Testing Self-Adaptive Systems: A Model-based Approach to Resilience

Testing Self-Adaptive Systems
A Model-based Approach to Resilience

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Georg Püschel
geboren am 15. Mai 1985 in Dippoldiswalde

Erster Gutachter: Prof. Dr.-Ing. Thomas Schlegel
Zweiter Gutachter: Univ.-Prof. Dr.-Ing. habil. Matthias Riebisch

Tag der Verteidigung: 05. Juni 2018

Dresden im Juni 2018

Statement of Authorship

I hereby certify that I have authored this thesis independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, June 28, 2018

Dipl.-Inf. Georg Püschel

Abstract

Autonomy is the most demanded yet hard-to-achieve feature of recent and future software systems. Self-driving cars, mail-delivering drones, automated guided vehicles in production sites, and housekeeping robots need to decide autonomously during most of their operation time. As soon as human intervention becomes necessary, the cost of ownership increases, and this must be avoided. Although the algorithms controlling autonomous systems become more and more intelligent, their hardest opponent is their inflexibility. The more environmental situations such a system is confronted with, the more complexity the control of the autonomous system will have to master. To cope with this challenge, engineers have approached a system design, which adopts feedback loops from nature. The resulting architectural principle, which they call self-adaptive systems, follows the idea of iteratively gathering sensor data, analyzing it, planning new adaptations of the system, and finally executing the plan. Often, adaptation means to alter the system setup, re-wire components, or even exchange control algorithms to keep meeting goals and requirements in the newly appeared situation.

Although self-adaptivity helps engineers to organize the vast amount of information in a self-deciding system, it remains hard to deal with the variety of contexts, which involve both environmental influences and knowledge about the system's internals. This challenge not only holds for the construction phase but also for verification and validation, including software test. To assure sufficient quality of a system, it must be tested under an enormous and, thus, unmanageable, number of different contextual situations and manual test-cases.

This thesis proposes a novel set of methods and model types, which help test engineers to specify precisely what they expect from a self-adaptive system under test. The formal nature of the introduced artifacts allows for automatically generating test-suites or running simulations in the loop so that a qualitative verdict on the system's correctness can be gained. Additional to these conceptional contributions, the thesis describes a model-based adaptivity test environment, which test engineers can use for testing actual self-adaptive systems. The implementation includes comprehensive tooling for creating the introduced types of models, generating test-cases, simulating them in the loop, automating tests, and reporting. Composing all enabling components for these tasks constitutes a reference architecture of integrated test environments for self-adaptive systems. We demonstrate the completeness and accuracy of the technical approach together with the underlying concepts by evaluating them in an experimental case study where an autonomous robot interacts with human co-workers.

In summary, this thesis proposes concepts for automatically and, thus, efficiently testing self-adaptive systems. The quality, which is fostered by this novel approach, is resilience: the ability of a system to maintain its promises while facing changing environments.

Publications

This doctoral thesis is based on the following peer-reviewed publications:

- Georg Püschel, Ronny Seiger, Thomas Schlegel (2012). *Test Modeling for Context-Aware Ubiquitous Applications with Feature Petri Nets.* In Modiquitous Workshop, pp. 1-4, 2012.
- Georg Püschel, *Testmodellierung für mobile Anwendungen.* In Proceedings of Innovationsforum Open4Innovation, pp. 65-69, 2012.
- Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Uwe Aßmann, *A Black Box Validation Strategy for Self-Adaptive Systems.* In Proceedings of ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, pp. 111-116, 2014.
- Georg Püschel, Sebastian Götz, Claas Wilke, Christian Piechnick, Uwe Aßmann. *Testing Self-Adaptive Software: Requirement Analysis and Solution Scheme.* International Journal on Advances in Software, ISSN 1942-2628, 7(1&2), pp. 88-100, 2014.
- Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Thomas Schlegel, Uwe Aßmann, *A Combined Simulation and Test Case Generation Strategy for Self-Adaptive Systems.* International Journal On Advances in Software, 7(3&4), pp. 686-696, 2014.

Further publications related to the thesis:

- Georg Püschel, *Test Modeling of Dynamic Variable Systems Using Feature Petri Nets.* Technische Universität Dresden, Fakultät Informatik. ISSN 1430-211X, TUD-Fl13-01-Sept. 2013. Technical Report, 2013.
- Georg Püschel, Christian Piechnick, Uwe Aßmann, *Generative und simulative Softwaretests für selbstadaptive, cyber-physikalische Systeme.* In Proceedings of the Multiconference Software Engineering and Management 2015, Koellen-Verlag, 2015.

Contents

1 Introduction **1**
- 1.1 Problem Description . . . 1
- 1.2 Overview of Adopted Methods . . . 3
- 1.3 Hypothesis and Main Contributions . . . 4
- 1.4 Organization of This Thesis . . . 5

I Foundations **7**

2 Background **9**
- 2.1 Self-adaptive Software and Autonomic Computing . . . 9
  - 2.1.1 Common Principles and Components of SAS . . . 10
  - 2.1.2 Concrete Implementations and Applications of SAS . . . 12
- 2.2 Model-based Testing . . . 13
  - 2.2.1 Testing for Dependability . . . 14
  - 2.2.2 The Basics of Testing . . . 15
  - 2.2.3 Automated Test Design . . . 18
- 2.3 Dynamic Variability Management . . . 22
  - 2.3.1 Software Product Lines . . . 23
  - 2.3.2 Dynamic Software Product Lines . . . 25

3 Related Work: Existing Research on Testing Self-Adaptive Systems **29**
- 3.1 Testing Context-Aware Applications . . . 30
- 3.2 The SimSOTA Project . . . 31
- 3.3 Dynamic Variability in Complex Adaptive Systems (DiVA) . . . 33
- 3.4 Other Early-Stage Research . . . 34
- 3.5 Taxonomy of Requirements of Model-based SAS Testing . . . 36

II Methods **39**

4 Model-driven SAS Testing **41**
- 4.1 Problem/Solution Fit . . . 41
- 4.2 Example: Surveillance Drone . . . 43
- 4.3 Concepts and Models for Testing Self-Adaptive Systems . . . 44
  - 4.3.1 Test Case Generation vs. Simulation in the Loop . . . 44
  - 4.3.2 Incremental Modeling Process . . . 45

4.3.3 Basic Representation Format: Petri Nets . . . 46
4.3.4 Context Variation . . . 50
4.3.5 Modeling Adaptive Behavior . . . 53
4.3.6 Dynamic Context Change . . . 57
4.3.7 Interfacing Context from Behavioral Representation . . . 62
4.3.8 Adaptation Mode Variation . . . 64
4.3.9 Context-Dependent Reconfiguration . . . 67
4.4 Adequacy Criteria for SAS Test Models . . . 71
4.5 Discussion on the Viability of the Employed Models . . . 71
4.6 Comparison to Related Work . . . 73
4.7 Summary and Discussion . . . 74

5 Model-based Adaptivity Test Environment 75
5.1 Technological Foundation . . . 76
5.2 MATE Base Components . . . 77
5.3 Metamodel Implementation . . . 78
5.3.1 Feature-based Variability Model . . . 79
5.3.2 Abstract and Concrete Syntax for Textual Notations . . . 80
5.3.3 Adaptive Petri Nets . . . 86
5.3.4 Stimulus and Reconfiguration Automata . . . 87
5.3.5 Test Suite and Report Model . . . 87
5.4 Test Generation Framework . . . 87
5.5 Test Automation Framework . . . 91
5.6 MATE Tooling and the SAS Test Process . . . 93
5.6.1 Test Modeling . . . 94
5.6.2 Test Case Generation . . . 95
5.6.3 Test Case Execution and Test Reporting . . . 96
5.6.4 Interactive Simulation Frontend . . . 96
5.7 Summary and Discussion . . . 97

III Evaluation 99

6 Experimental Study: Self-Adaptive Co-Working Robots 101
6.1 Robot Teaching and Co-Working with WEIR . . . 103
6.1.1 WEIR Hardware Components . . . 104
6.1.2 WEIR Software Infrastructure . . . 105
6.1.3 KUKA LBR iiwa as WEIR Manipulator . . . 106
6.1.4 Self-Adaptation Capabilities of WEIR . . . 107
6.2 Cinderella as Testable Co-Working Application . . . 109
6.2.1 Cinderella Setup and Basic Functionality . . . 109
6.2.2 Co-Working with Cinderella . . . 110
6.3 Testing Cinderella with MATE . . . 112
6.3.1 Automating Test Execution . . . 112
6.3.2 Modeling Cinderella in MATE . . . 113
6.3.3 Testing Cinderella in the Loop . . . 121
6.4 Evaluation Verdict and Summary . . . 123

7 Summary and Discussion 125
7.1 Summary of Contributions . . . 126
7.2 Open Research Questions . . . 127

Bibliography **129**

Appendices **137**

Appendix Cinderella Definitions **139**
1 Cinderella Adaptation Bounds 139
2 Cinderella Self-adaptive Workflow 140

List of Figures

1.1 The state space of an SAS . . . 2
1.2 Thesis outline . . . 5

2.1 The control loop structure of SAS [IBM06] . . . 11
2.2 The causality chain of faults, errors, and failures (redrawn from [ALRL04]) . . . 14
2.3 The V-Modell (redrawn from [Gre10]) . . . 16
2.4 Dynamic test process (redrawn from [IEE13a]) . . . 16
2.5 The MBT Process [UL07] . . . 19
2.6 Sample finite state machine for a simple webshop . . . 22
2.7 Example of the original feature tree notation (taken from [KCH+90]) . . . 24

3.1 Taxonomy of requirements of model-based SAS testing. . . . 36

4.1 Problem/Solution fit . . . 42
4.2 The conceptional process of SAS testing . . . 45
4.3 Petri net for the drone example . . . 47
4.4 Petri net reachability graph . . . 48
4.5 Usage of the Petri net in simulation and test-case generation . . . 49
4.6 Context variability model . . . 52
4.7 Adaptive Petri net modeling adaptive behavior . . . 54
4.8 Usage of the variability model in simulation and test-case generation . . . 56
4.9 Different types of change models. . . . 58
4.10 Variant manipulation and its effect on the definition of expected behavior . . . 59
4.11 Virtually timed change models . . . 60
4.12 Integration of stimulus models in simulation and generation . . . 62
4.13 Adaptive Petri net with timer transitions . . . 63
4.14 Generation and simulation infrastructure with timer-transition-based synchronization . . . 65
4.15 Explicit representation of adaptation modes . . . 66
4.16 Stimulus models with event production . . . 69
4.17 Causal chain of stimulus and reconfiguration . . . 69
4.18 Generation and simulation infrastructure including reconfiguration and explicit adaptation modes . . . 70
4.19 Comparison of the proposed approach to related work . . . 73

5.1 Layers and components of MATE . . . 77
5.2 Packages of the MATE metamodel with dependencies . . . 79
5.3 Feature-based variability metamodel (package `features`) . . . 80

5.4 Classes and relations of the abstract syntax for variability constraints (package `constraints`) . . . 81
5.5 Abstract syntax of functions language (package `functions`) . . . 82
5.6 The abstract syntax of the term language (package `terms`) . . . 84
5.7 The abstract syntax of the test action language (package `actions`) . . . 85
5.8 Metamodel for adaptive Petri nets (package `petri`) . . . 86
5.9 Common metamodel for automaton-based stimulus models reconfiguration automata (package `reconfiguration`) . . . 88
5.10 Metamodel for test-suites (package `test`) . . . 88
5.11 Classes and relations of the generator framework (package `generator`) . . . 89
5.12 Metamodel for test-automation (package `automation`) . . . 92
5.13 The process of SAS testing with MATE . . . 94
5.14 A running simulation in MATEs . . . 96

6.1 Functional modes of the DLR co-worker, redrawn from [HSF+11] . . . 102
6.2 WEIR hardware . . . 104
6.3 WEIR software components . . . 105
6.4 WEIR-controlled KUKA LBR iiwa . . . 107
6.5 Adaptation bounds, movables, and state machine for collision detection . . . 108
6.6 Example WEIR adaptive workflow . . . 109
6.7 Cinderella setup . . . 111
6.8 Cinderella co-working behavior . . . 111
6.9 Variability model for Cinderella . . . 113
6.10 Event flow through the Cinderella test model . . . 114
6.11 Spatial context stimulus model for Cinderella . . . 115
6.12 Model of movement of body parts for Cinderella . . . 116
6.13 Reconfiguration automaton for one Cinderella picking box . . . 118
6.14 Cobotics reconfiguration automaton for Cinderella . . . 119
6.15 Adaptive Petri net for Cinderella . . . 120
6.16 Running Cinderella test . . . 122
6.17 Representation of proposed concepts in Cinderella . . . 123

1. Introduction

Imagine robots that build our houses, organize our household, and manage our everyday problems. Others transport us to our job or holiday location, or wherever we need to go. Such systems, however, are not anymore bound to a single, controlled location, such as industry workstations or a living room's carpet. Consequently, they have to cope with the dynamics of real environments, just like humans do. Such intelligence requires a new kind of software to take control and to determine at run-time how the whole system should *adapt* to newly explored surroundings. Software engineering characterizes such kind of system as *self-adaptive*.

Self-adaptive software or—to a greater extent—a self-adaptive system (SAS) observes properties of itself and its environment—or more precisely its *context*—and automatically adapts itself at run-time to a set of goals [Lad97]. The loop-wise execution of this process allows SAS for autonomously adhering to the goals, that is without manual intervention. Vice versa, autonomy in physical, mostly uncontrollable environments demands such a self-adaptive mechanism. Recent and future cyber-physical systems, self-steering cars, unmanned air vehicles, and many other applications of autonomy will benefit from recent findings of SAS research.

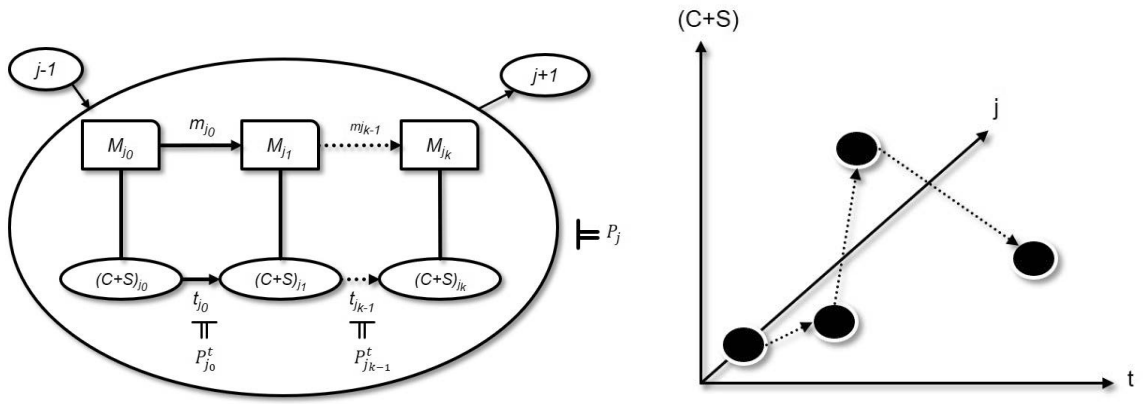
From the perspective of quality assurance, failures in self-adaptive applications can be highly critical. Self-adaptivity unavoidably leads to systems that interact autonomously with other systems or people and their property with the risk of severely damaging them. Thus, the functional correctness and the quality of SAS has to be verified and validated extensively before delivery.

Software engineering processes employ validation and verification (V&V) on different abstraction layers and at different points in time during the product lifecycle. Testing, as the most widely adopted method of V&V, is based on code, components, interfaces, and requirement specifications, each with increasing level of abstraction. This thesis focuses on testing the functional correctness of SAS, performed on the abstract level of interfaces and requirements, which we call *black-box testing* because it neglects internal structures of the system under test (SUT).

1.1. Problem Description

Despite the severe need to test SAS, their complexity does not comply well with traditional test approaches. Changes in the environment, in the adaptation of behavior, and the adaptation of the SUT's structural appearance, as well as the causality in between, are not considered in standard testing. Jean-Claude Laprie summarizes the quality of SAS to be dependent on external influences as resilience:

> **Definition 1 (Resilience)** "The persistence of service delivery that can justifiably be trusted, when facing changes." [Lap08] □



(a) Context states (C) combine with system states (S). The resulting state must adhere to models M (redrawn from [CdLG+09]).

(b) The behavior additionally depends on history. Hence, a third time dimension t is considered.

Figure 1.1.: The state space of an SAS.

Testing SAS is challenging and unique because the bandwidth of conditions in context and adaptation generates structural and behavioral variability that by far exceeds those of traditional systems. This variability can be described as a highly complex state space, as already recognized by Cheng et al. [CdLG+09]. According to them, each state is a compound of context and system conditions. When one of the two changes, a completely new situation constitutes. This relation is illustrated in Figure 1.1a. The illustration shows the configuration spaces of the context C and the system S, 1which are combined in sense of the propositions that hold at a certain point in time. Having 100 context states and 100 system states gives $100 * 100 = 10000$ possible configurations. Additionally, the system adapts itself or, in other words, it changes its operation mode j. Thus, adaptation further inflates the state space. Of course, the specification does not allow all of those combinations; the state space is segregated in a valid and an invalid subset. V&V is in charge to assure that the risk of trespassing this line of segregation remains low during the productive operation phase.

The specification of permitted behavior defines properties $P^t_{j_i}$, which can be tested for certain transitions and others P_j that are expected to hold for a complete adaptation mode. All these expectations must be reflected by models for states M_{j_i} and models for transitions m_{j_i}.

In black-box testing—as investigated in this thesis—, another effect is the incomplete knowledge of the inner processes of the SUT. Testers can never be sure about eventually changed states so that only a strict history of sequences actions guarantees the reproduction of a targeted state. This effect is illustrated in Figure 1.1b. Not only context, system, and adaptation mode influence the assumed state but also the history of performed test actions, which here is represented by the time dimension t.

To consider all these effects, validation methods usually implement three different mechanisms for checking given expectations: (1) a mechanism to enforce a particular state, (2) an oracle that anticipates the expected properties, and (3) a mechanism for comparing those expected properties with measured ones. Standard test-cases incorporate all three—they enforce a state based on action sequences and test data, query for results and compare them to given values defined in a test data repository. However, in traditional testing, the tester manually provides all these ingredients, which is inefficient in the face of an SAS' highly variable behavior. The combinatorial effects cause *test-case explosion*, which is the multiplication of the number of test-cases caused by each additional variable to be considered. Similarly, the notion of *state space explosion* describes the combinatorial effects on the operational and parameter space, which must be covered by those test-cases. Whereas test-case and state space explosion are already hard to manage when testing traditional systems, testers are lost in SAS testing.

Another misconception is the assumption that all states of an SAS under test (SASuT) are explicitly known to designers and testers. The autonomy fostered by SAS also requires an algorithmic design in which reconfiguration is automatically decided based on rules, heuristic methods, or even artificial intelligence, including machine learning approaches. Similarly, the physical environment of autonomous applications cannot be controlled with the same precision as for virtual environments. Instead, the effects of physical manipulation can often only be measured and the system's reaction be verified based on this measurement. Here, so-called stigmergy (i.e., information transfer by physical manipulation [KFH15]) and external environment dynamics affect the real context state. Consequently, manual testing faces the problem that the test designer is not able to explicitly pre-define all states that will be reached during processing. Thus, much more information is implicit and must be investigated and checked with the means of software testing.

Some past research proposed to built-in mechanisms of self-testing into SAS [KACC11]. In this approach, components of SAS are equipped with test-cases or templates that are executed at adaptation time. Such mechanisms even allow for testing unanticipated change where components are integrated at run-time without that a designer could have anticipated the reached configuration. This thesis neglects such methods as they fall short in classical software quality assurance. In contrast, products have to be provided in an acceptable quality at the moment when they are shipped to the customer.

In consequence, the manufacturer must perform tests in advance and systematically so that engineers can subordinate SAS testing to the standard processes of V&V. However, due to the lack of sufficient SAS test methods, no effective tooling yet exists. Such tooling should incorporate support for all test process steps: test design, automation, execution, and reporting. To cope with the special requirements of SAS, each of these steps must be enriched with new technical concepts.

In summary, the concrete vision of this thesis is an efficient set of models and methods for systematic a-priori testing SAS together with a ready-to-use realization. Consequently, the following major problems arise:

(P1) **Vast expansion of the state space of SAS:** SAS monitor changing properties of their context and react at run-time by changing their configuration in the form of structural composition, parameterization, and service behavior. To assure the system's quality adequately, a wide variety of contextual situations have to be enforced and the reaction verified. Furthermore, not only the current situation may play a role but the past situations as well. All these factors introduce multiply variation points, which produce a much larger state space than in traditional computing. In consequence, overlooking this state space and testing paths through it for verification is highly demanding for test experts.

(P2) **Missing tooling for testing SAS:** Test engineers lack an integrated environment, which allows for performing tests on SAS. This lack includes tools for test design, automation, execution, and reporting. Respective solutions should incorporate support for all these process steps and leverage standards of test tooling to SAS-specific requirements.

1.2. Overview of Adopted Methods

The problems enumerated in Section 1.1 shall be tackled by different concepts, which combine to an overall conceptional and technical solution. The foundations of this solution are outlined in the following.

Model-based Testing The necessity of abstraction of context and system behavior by a validation model was already recognized by Cheng et al. in [CdLG+09]. Against this background and with the focus on testing, the foundation of this thesis is model-based testing (MBT). MBT puts

models in place of test-cases; more precisely, the first generates the latter [UL07]. Thus, a test engineer no longer writes test-cases as sequences of actions and assertions but instead defines a formal model from which test-cases are produced automatically. A test model defines structural and behavioral expectations to the SASuT and is searched for sequences that enforce certain system states. Also, the model specifies properties for each state, which are then verified against measurements from the real SASuT. Thus, testing of systems with a lot of variations gets much more efficient because testers are freed from the need to specify a high number of test-cases by hand.

Dynamic Variability Modeling State-of-the-art MBT applies to standard, non-adaptive systems. To leverage MBT methodology for SAS, it must be extended for the problems that are specific to self-adaptation. As introduced in Problem (P1), new dimensions of variability arise from run-time adaptation to dynamic environments. Dynamic change in properties of the monitored environment and the SASuT itself are causally connected, which has to be tackled by novel MBT methods. Furthermore, both context and the resulting adaptation depends on the history of the system. To tackle these challenges, this thesis integrates MBT with dynamic variability modeling. The latter provides means to abstract and organize commonalities and differences of configurations of both context and SAS. For representing the dynamics and the causality between context and system variability, re-configuration at run-time shall be described by dynamic variability models.

Duality of Test Generation and Simulation Almost all concepts that are proposed in this thesis can be used within a classic MBT process and simulative validation. Classic MBT means to generate a fixed set of test-cases from the model and later execute the generated sequential test-cases. In contrast, simulative validation executes the model directly while applying the modeled actions on a real parallel-running SASuT, which is further observed and its state and outputs verified against the model. Such an approach is often referred to as "in the loop" (ITL) simulation and is beneficial in situations where certain situations cannot be modeled with sufficient precision. As the decision logic, which is resolved during test generation, is still accessible in simulation, observed environment data can be embedded in the model. Such an update mechanism was already claimed to be beneficial by Cheng et al.:

> "On one side the model must be efficiently updated to reflect the system changes, on the other it should still reflect an accurate representation of reality"
>
> [CdLG+09, p. 19]

Only based on the updated information from ITL observations, test oracle's can generate correct expectations. Both test generation and simulation shall be performed on almost the same models and tools to gain more reuse.

1.3. Hypothesis and Main Contributions

This thesis employs a model-based approach to cope with conceptional problems arising from a complex state space and implicit information within the tested black-box. The approach claims to improve SAS' testability and that the used models provide appropriate expressiveness to enforce and verify parts of the state space, which could only be tested before with enormous effort. Thus, the following hypothesis can be stated:

> The proposed conceptual and formal models, together with the process that puts those models in use, solve the automation of testing the dynamics of self-adaptive systems to a novel extent of the contextual and behavioral state space.

This hypothesis promises certain contributions that reach from conceptional approaches to implementations:

> (C1) **Model-driven methods** for testing before non-covered regions of an SAS' state space based on generated test-cases or simulation in the loop
>
> (C2) A **comprehensive formalization** in the form of metamodels that implement all aspects of the SAS test methods
>
> (C3) A reference architecture for an **integrated test environment** (MATE) that realizes the proposed models and allows for employing them along a standard dynamic test process

Besides these main contributions, several minor ones are elaborated during the chapters ahead. We list them in the thesis' summary where they can be understood in detail.

Based on MATE, a later evaluation shall demonstrate the correctness of the hypothesis as well as the adequacy of the above-listed contributions. Hence, an experimental study has been performed and documented in the evaluation part of this thesis.

1.4. Organization of This Thesis

The thesis' parts are outlined in Figure 1.2. Part I introduces several research domains, which created the foundations this thesis' contributions shall rely on. First, the background is discussed and, second, related work is presented. In Part II, the modeling concepts are described, including examples and formalization. Subsequently, MATE is presented as an implementation of the introduced models and the dynamic test process for SAS. In Part III, an experimental study is elaborated, which validates the proposed concepts in a robotic scenario. Finally, a summarization of this thesis, including open research questions, follows.

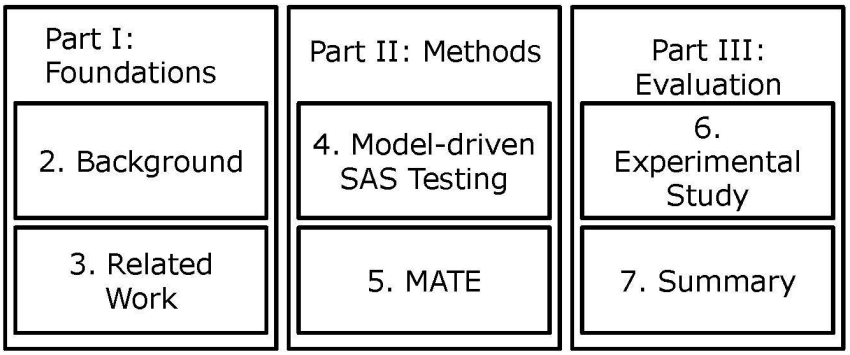


Figure 1.2.: Thesis outline.

Part I.

Foundations

2. Background

As argued in this thesis, Model-based testing (MBT) of SAS relies on a tremendous body of knowledge in numerous research directions. Not only that SAS has been identified in separate communities as the next generation of software system., Even more, the bandwidth of solutions that claim to solve SAS-inherent challenges is quite heterogeneous.

Similarly, in test research, several model-based techniques have been found. Some approaches generate from design artifacts; others introduce entirely new formalisms and notations for testing. The level of expressiveness of an MBT concept determines the level the automation of test design, which this thesis aims to improve with the particular focus on SAS.

As briefly discussed in Section 1.2, variability modeling is an appropriate match to connect both SAS and MBT. Research on software variability evolved primarily in the field of software product line engineering (SPLE), which is a necessary preliminary for understanding the motivation behind variability management. For at-run-time adaptive systems, the SPLE community developed the concept of dynamic variability modeling, which shall be adopted in this thesis.

Because of this bandwidth of background topics, this chapter follows coarse-granular segmentation of the introduced body of knowledge. The theoretical foundations of self-adaptivity are presented in Section 2.1, model-based testing in Section 2.2, and dynamic variability management in Section 2.3. Commonly accepted definitions are recited as well as related work that establishes a basis for comprehending the contributions of this thesis.

2.1. Self-adaptive Software and Autonomic Computing

The history of SAS research is brief in comparison to the time span researchers spend with software engineering. As the term *self-adaptivity* suggests, there are two crucial factors that make out an SAS: firstly, it provides mechanisms to adapt itself and, secondly, it decides automatically to adapt at run-time. Software is a formal artifact and, thus, adaptation decisions rely on objective criteria—for instance on an observed quality, which is guaranteed by the system and drops below a given threshold. To navigate itself out of this situation, a system should be *self-aware*, which is a necessary precondition for the decision making in SAS.

Before the background of above intuitive understandings of SAS, software researchers have outlined the notion of SAS more precisely. One of the first definitions by DARPA engineer Robert Laddaga circumscribed SAS as follows:

> "Self Adaptive Software evaluates its behavior and change behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible." [Lad97]

Furthermore, Laddaga explicitly distinguishes SAS from systems explicitly relying on neural networks or genetic programming, which he sees as potential reasoning techniques. Instead, he aims to build systems that explicitly know about their structure, functionality, and performance and automatically reason on this knowledge with the goal to act more intelligently.

Later, in 1999, Oreizy et al. recognized the strong relationship between SAS and autonomic computing [OGT+99]. The authors derived the demand for SAS from the necessity to reuse architectural components in different applications. As a solution to this requirement, they proposed automated reasoning and adaptation. From their perspective, the main challenges of SAS engineering are planning, coordination, monitoring, evaluation, and the implementation of seamless adaptation by means of software architecture.

Autonomic computing as distinct research domain gained interest in software engineering since IBM claimed several challenges and proposed respective architectural components after the millennium [Hor01][KC03][IBM06]. The IBM engineers' initial desire was to shift responsibilities in managing complex heterogeneous systems from humans to automated solutions. According to them, creators of business applications face a complexity crisis. They have to integrate various heterogeneous, often networked components that fulfill a common task. Test concerning configuration, deployment, and maintenance can no longer be understood and managed by humans effectively and efficiently. Additionally, modern business applications evolve constantly while they are productively used. These factors cause the total cost of ownership (TCO) to increase, which has to be counteracted.

To remedy these problems, management tasks should be delegated to automated decision mechanisms. Thus, the human administrators can better focus on problems with a higher value to the actual business. Low-value tasks are performed by intelligent computing systems so that the TCO is decreased.

The task delegation is performed during run-time by administrators. In this process, the task has to be formulated as a policy, which is assembled from goals that define what the automated system is expected to achieve and constraints that restrict the means the system is permitted to use. After configuring the system with goals and constraints, the autonomic computing system has to compute necessary actions on its own (i.e., autonomic). The process runs periodically in a *control loop*, which reasons on input and decides adaptations.

Both bodies of knowledge developed in SAS research and autonomic computing intersect strongly, whereas the application domains are quite different. At DARPA, typically military applications were considered, including automated target recognition, signal and image processing, and robotics. Besides such military domains, SAS plays a vital role in civil applications like autonomous cars, drones, and home robotics. In contrast, autonomous computing focused on improving processes in business applications and automating periodically recurring tasks of such software.

2.1.1. Common Principles and Components of SAS

IBM engineers Jeffrey O. Kephart and David M. Chess encouraged their vision [KC03] of self-adaptive and autonomic computing and published ideas about concepts and principles to be standardized in the form of an architectural blueprint [IBM06]. In this section, starting with IBM's notion, the commonly accepted principles and components of SAS and autonomic computing are presented. Thus, a common understanding of self-adaptivity shall be reached that is the premise of the generality of the proposed test approach.

Concerns of Adaptation: Self-* Properties

SAS and autonomic computing employ control loops that collect environment observations. Based on the gathered data, quality measures are investigated and reasoned about to derive

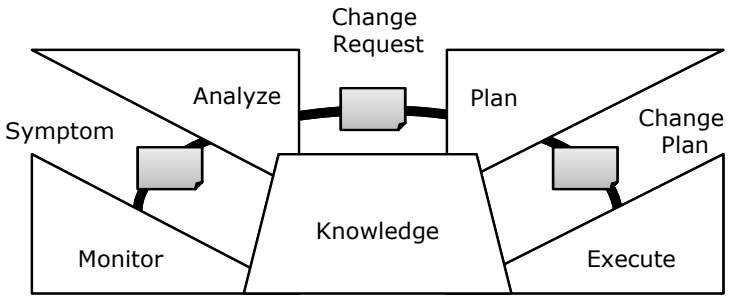


Figure 2.1.: The control loop structure of SAS [IBM06]. *Sensor and otherwise input is monitored and fused to symptoms, which later are analyzed for the need of change. Potential change requests are computed in the plan phase resulting in a change plan, which is finally executed. Required information to reason about is stored to and taken from knowledge sources.*

adaptation actions. Several qualities can be considered during the adaptation process. In [KC03], four different categories of control loops are distinguished:

- **Self-configuring:** Adapt dynamically at run-time to changes of the environment using policies that are provided by IT professionals. Thus, a continuous operation should be ensured.
- **Self-healing:** The system uses correction policies to react to detected malfunctions. Thus, the system's resilience is improved.
- **Self-optimizing:** Tune the system's resources in response to dynamically changing workloads. The quality of service is steered towards the requested requirements of the users.
- **Self-protecting:** Find behaviors that threaten the system's security and privacy and react by corrective actions. Thus, the security and privacy policies are enforced even during attacks.

All categories of control loops have a common structure originating from business processes for the control of incident, problem, and change management. As each role of such a process has power on a specific scope, in an autonomic system an autonomic manager controls a particular scope as well. Therefore, the system must be able to automatically configure, heal, optimize, and protect itself using a set of knowledge sources.

Later, the four types of control loops were adopted as *self-* properties*. In [ST09], Salehie and Tavildari classify these properties as major levels of adaptation and hierarchically arrange them below the general level of *self-adaptiveness*. Furthermore, *self-awareness* and *context-awareness* constitute a most primitive layer in this hierarchy.

Process of Adaptation: Components and Feedback Loops

The named blueprint [IBM06] also proposes a standard architecture for autonomic computing. On the lowest level, several hardware (e.g., CPU, storage) and software (e.g., databases, services) *resources* are situated. The resource management is performed using standard interfaces called *touchpoints*, managers, and orchestration of these independent agent-like constituents.

Adaptation mangers of different scope use the concept of control loops, which are structured as depicted in Fig. 2.1. The first function is *Monitor* by which details on the managed resources are collected. The gathered information is aggregated, correlated, or filtered. The result is a set of *symptoms*, which are handed over to the *Analyze* function. The latter function is in charge to perform a data analysis (e.g., prediction) and determine whether a change is required.

For instance, policies are used for this purpose and matched with the observed situation. The analysis may result in a change request. The *Plan* component derives *change plans*, which comprise actions that define the reorganization tasks for fulfilling the new policy. There may be only a single action or even a complex workflow necessary to alter the system. These actions are scheduled and performed by the *Execute* function.

The complete control loop is supported by a *knowledge base*. Knowledge can be provided by *knowledge sources* of different types. Firstly, the policy is passed directly to an autonomic manager and is stored as another element of the knowledge base. Secondly, external sources can support the decision making. For instance, symptom descriptions or historical data can be stored to analyze later and compare them with recently monitored events. The third type of knowledge sources encompasses information that is created by the autonomic manager itself. Thus, monitored sensor data, created plans, and execution results can be stored. For this purpose, each function may also have the ability to alter stored knowledge.

Loop-wise computation of observed data and derivation of adaptation decisions is the most referred concept of SAS and incorporates much variability, which was investigated by Andersson et al. in [ADLMW09]. The authors enumerate a multitude of dimensions of different categories. These dimensions include properties and qualities of goals to achieve, the change that causes adaptation, and mechanisms to adapt and effects adaptation. Along the process, a control loop runs, which instantiates all these parameters.

2.1.2. Concrete Implementations and Applications of SAS

For all adaptation concerns, a multitude of SAS frameworks has been implemented. A good overview is given by Salehie and Tahvildari in [ST09]. Whereas researchers mostly aimed at developing generic approaches for SAS, such as architectural frameworks, engineers implement solutions to concrete domain problems. For both directions, several works are discussed in the following.

From an architectural perspective, different topological variants have been proposed. For instance, IBM's blueprint describes a strongly layered architecture for SAS [IBM06]. On the lowest layer, resources, such as servers, storage, and network, are managed. On the next level, each resource is instrumented by a so-called *touchpoint*, which is a uniform resource management interface. Furthermore, touchpoints are controlled by autonomic managers, which run a local control loop. Globally, touchpoint managers are orchestrated by more generic autonomic managers. On the highest level, manual intervention is possible for tasks that were not automated yet. IBM's blueprint can be understood as a reference architecture for SAS, and many later proposals refer to it as such.

Alternatively, Garlan et al. constructed their Rainbow system, which performs an architecture-based self-adaptation approach [GCH+04][GSC09]. In Rainbow, a model of the system's architecture is maintained at run-time on an architecture layer, whereas resources, effectors, and probes for measurement are bundled on a separate system layer. Both layers are connected via a translation infrastructure, which mediates information exchange. Based on the architectural model, reasoning is performed to determine when and how to adapt.

For the evaluation of the Rainbow framework, the authors introduced the self-adaptive web framework ZNN.com, which provides news and adapts to load spikes [ASP13b]. The goal quality is response time, which may become unacceptable due to high request frequency. In reaction, the system adapts by enlisting new servers or switching to text-only content delivery. Also, adaptation strategy considers user-defined goals, such as costs to be minimized or prioritization of the user experience of multimedia content before response time.

In contrast to these layered architectures, Raibulet et al. propose in [RAM+06] to separate a networked system into views for resource structures, the topology of connections, and resource location. Based on these views, the system reflects itself and knowing the quality of service (QoS) that can be delivered in a specific setting. For each query to the service, the required QoS

is evaluated so that necessary adaptations can be derived. The concrete adaptation strategy strongly depends on the application domain so that only an abstract framework for strategy definition is provided.

Furthermore, a relevant project is Managing Distributed Adaptation of Mobile Applications (MADAM [AHPE07]), where adaptation decisions are based on utility functions on provided and required QoS properties, which are annotated to components. MADAM runs a control loop that monitors environment properties and manages adaptation autonomously.

The Autonomia framework by Al-Nashif et al. provides another elaborate approach for constructing SAS [ANKH+08]. The proposed architecture comprises two components: (1) the component run-time managers (CRM), which actively executes the control loop, and (2) the component management interface (CMI), which allows for defining adaptation policies and other to necessary specifications. Further, the CMI connects to sensors and actuators as well as rules that define adaptation actions. Autonomia also implements the orchestration aspect as discussed by the above mentioned IBM blueprint in the form of compound components and CRMs. The authors present a multitude of applications in the domains of consumer, military, financial, and scientific computing. Especially grid computing and network security (e.g., intrusion detection) have been tackled in their work.

Alternatively, peer-to-peer topologies are constituted of widely independent entities. Adaptation in such decentralized systems can be enabled for instance by the architecture proposed by Baresi et al. in [BGT08]. Here, supervised elements (SE, i.e., sensor and actuators) and supervisors are distinguished. Supervisors run a subscription/notification protocol that they can be connected to the supervised elements of interest. Furthermore, the supervisors monitor their subscriptions and run adaptation of different granularity (in a federation of supervisors or clusters of SEs). One remarkable specialty of this approach is that supervisors are connected to SE via aspect-oriented programming (AOP) because adaptation is considered a cross-cutting concern.

Engineers apply SAS principles in situations where autonomy is needed to decrease manual workload and, thus, the TCO, or to cope with many influence factors at run-time. For the aeronautics domain, Mahadevan et al. built an SAS that monitors the health state of the air data inertial reference unit (ADIRU) and adapts by performing mitigation actions [MDK11]. The authors call their approach software health management and classify it as advanced fault tolerance technique.

Self-organization is frequently desired in robotic cooperative scenarios. An instance of such a system was constructed by Zhong et al. in [ZD11]. The authors employ a model of goals, capabilities, and roles. Without using explicit reorganization rules, the system can assign robots with appropriate capabilities to roles that are necessary to reach a certain goal. Another self-organizing robotic scenario is shown in [NG15] by Niemczyk and Gheis. Here, not the actual task of a robot entity but information sharing is focused. The approach is rather generic and allows developers for creating robot teams with cooperative information processing, which is adapted to optimize qualities like transfer time.

In summary, most implementations of SAS principles are, on the one hand, academic but, on the other hand, quite heterogeneous in the appearance, nomenclature, and employed techniques. However, each of them employs a control loop as the central mechanism, which is, therefore, the unifying element in SAS engineering and, consequently, must be taken into account in this thesis' efforts to tackle the SAS test challenge.

2.2. Model-based Testing

Dependable software requires a proper application of quality assurance methods during planning, development, and use. Testers are in charge to ensure that customers experience a flawless product after deployment. Otherwise, post-delivery failures are not only very costly and effect

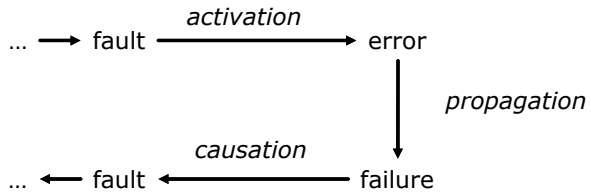


Figure 2.2.: The causality chain of faults, errors, and failures (redrawn from [ALRL04]).

the image of the software manufacturer, but also potentially cause severe damage to people or property. Thus, software and, more general, systems have to be tested exhaustively before delivery according to a hopefully correct requirements specification.

2.2.1. Testing for Dependability

Testing is the most-adopted method of fault avoidance. On a more general level, fault avoidance is a category of means for dependability besides fault tolerance, fault removal, and fault forecasting. In [ALRL04], Avižienis et al. give a broad overview of failure-related glossary and definitions in the field of dependent information systems. These notions shall be discussed in the following.

Information systems consist of a set of interacting entities, of which some may be designed, and others are established at run-time. For instance, a search engine comprises components, such as crawlers, a web interface, a database, and an indexing mechanism, but all data is gained later, after development. During actions for fault avoidance, only the search engine itself, but not all the searched data, can be considered. However, the system is expected to cope with potentially harmful influences from such external sources, which cannot be fixed by quality assurance. The frontier between the environment and the system is defined as *system boundary*, which has to be adequately designed by engineers.

Each system has a certain *function*, which is defined by a *functional specification* comprising definitions of the system's functionality and performance. To implement this specification, a system has a certain *behavior*. Behavior can be described by a *sequence of states*. Whereas single components may span isolated state spaces, the *total state* of the system comprises the complete set of states of computation, communication, stored information, interconnections, and physical conditions.

Furthermore, information systems only make sense when they interact with a human being or a technical counterpart. The interaction is provided as a *service*, which is the behavior that is perceived by a user. The user may not only be a human but also another system that receives the service from its provider.

The service is delivered at a *service interface*. Due to potential information hiding at this interface, the counterpart of the service only perceives a limited behavior viewed as *external states*, whereas *internal states* cannot be observed directly. Finding or avoiding malfunctions during the service delivery is the crucial interest of quality assurance. Such a malfunction can be defined as an event of deviation from the correct service.

A *service failure*, abbreviated *failure*, is caused by flaws during development, interaction, or due to physical conditions. The user perceives the failure as a deviation of the observed external state from the expectations that are defined in the system's specification. The failure-underlying state is called *error* and its cause *fault*. In summary, faults, errors, and failures span a causal chain. As depicted in Figure 2.2, a fault can *activate* an error, which further can *propagate* a failure. The transition from failure to another system part's fault is called *causation*. It must be mentioned, that the error activation, in particular, is optional. A fault may also be *dormant* so that no errors are produced because the functionality is not used or the fault is covered by a compensating part of the system.

As means for fault avoidance, testing checks information systems for the existence of failures. While debugging, an engineer investigates the error at the moment of failure occurrence and tries to identify the fault. However, due to the step-wise construction of software systems from requirements along design to implementation, the search for failures must be performed on different levels of abstraction. To improve the understanding of these different perspectives, the next section covers aspects of the test process and test methodology.

2.2.2. The Basics of Testing

Testing can be performed for an integrated software system or just some of its parts. The particular approach depends on the point in the software life cycle, the availability of interfaces and completed components, and the level of abstraction to be considered. A canonical impression on the relation between abstraction, construction step, and test methodology is delivered by the German V-Modell, which is discussed in the following section.

V-Modell: Levels of Testing

Depending on the knowledge, interfaces, and budget a tester is equipped with, testing can be performed on different levels of abstraction. The more detailed single components are investigated, the more precisely their preliminaries have to be specified. As software engineering processes target on refining initial knowledge over time, potential verification methods do as well. For instance, the specification of the German *V-Modell* [Gre10, p. 375] includes a visualization of relations between testing and design, as depicted in Figure 2.3. The process starts from the left upper corner of the `V` with an exploratory collection of information on the system to be developed. After a certain time, organizational plans and requirements can be specified. In the following phases, the system is constructed according to these requirements with an increasing level of detail. The process starts with a rather abstract product design, which is followed by a more detailed design and later implementation of code. After the final implementation step is concluded, each level has to be verified or validated based on the artifacts that were created during the design phase on the respective level of abstraction. In contrast to the definition of verification and validation in the context of dependable systems (cf., Section 2.2.1), verification is understood as *building the system conforming to the specifications of the previous phase* and validation as *building the system appropriate to its indicated purpose*. Thus, verification takes place on the lower levels, where formal or informal design specifications are available; validation on the upper levels, where the system can be checked against a user's needs.

The most implementation-near test level is *unit testing* where single modules of the system's code are directly stressed with input data and checked against pre-defined output data. Afterwards, the modules and components are increasingly integrated and tested for correct interaction and performance (*integration test*). The complete integrated system can be analyzed in the *system test* phase. An examination whether the constructed product matches its desired design is done during the *acceptance test* phase. The conformance of the system is checked above the baseline of requirements and can be finished by concept validation.

Whereas the V-Modell includes a description of test activities, a more focused model was launched by ISO, IEC, and IEEE and standardized under the reference 29119. It describes the test process as an isolated engineering discipline.

A Standard Process of Software Testing: ISO/IEC/IEEE 29119

> "It is generally accepted that it is not possible to create perfect software." [IEE13a]

Besides its role in the V-Modell, testing is essential to all accepted software development life cycles. To unify the understanding of test concepts, software testing has been standardized in

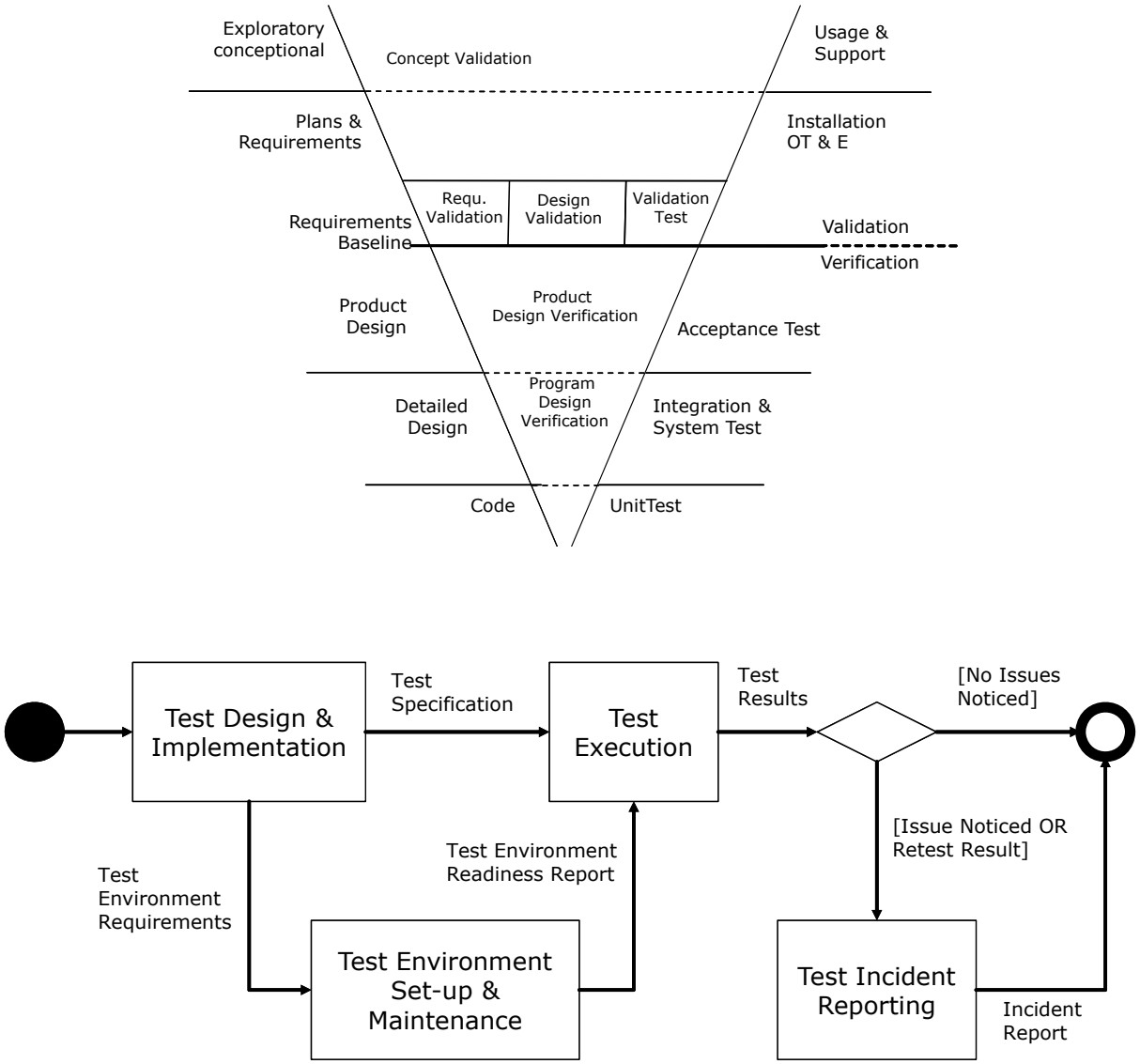


Figure 2.4.: Dynamic test process (redrawn from [IEE13a]).

ISO/IEC/IEEE 29119. The standard is organized along three aspects of testing, which have been published yet: (1) concepts and definitions [IEE13a], (2) test processes [IEE13b], and (3) test documentation [IEE13c]. The most significant statements of ISO/IEC/IEEE 29119 are discussed in this section for later reference.

The standard proposes to separate the process of testing in three sub-processes.

1. **Organizational Test Process:** The highest level of organizational test management includes test specifications, such as an Organizational Test Policy and Organizational Test Strategy.

2. **Test Management Process:** Test management including project management (planning, monitoring, and controlling) and type test management.

3. **Dynamic Test Process:** Lowest-level test activities like designing and implementing tests, executing them, and reporting the outcomes.

All three sub-processes interact by the interchange of certain artifacts, such as policies and specifications. Depending on the type of testing, artifacts from requirement engineering (ac-

ceptance testing), architectural design (system and integration testing), and from the detailed design (unit testing) are processed.

As this thesis discusses a specific test practice—model-based testing—for a specific category of test objects—SAS—, the dynamic test process is of particular importance. Activities and communicated artifacts of the dynamic test process are depicted in Figure 2.4. The first activity *test design & implementation* aims at the creation of test specification consisting of test-cases and test procedures. During this sub-process, it is also necessary to specify test coverage measures and test completion criteria. Furthermore, it may be the case that the specification activity has to be re-entered due to re-plan or to identify additional test conditions.

Besides the test specification, in test design, requirements of the test environment are derived. These requirements constitute a set of conditions that the test environment must conform to. The subsequent activity *test environment setup & maintenance* involves building the environment according to the conditions and maintaining it. Here, software and hardware are assembled appropriately so that test-cases can be run and test data can be applied as defined in the test specification. The test environment readiness report contains all information on how this environment can be used and configured.

During *test execution*, test-cases are assembled in the test specification and run in the test environment. Running means applying the actions of the test-cases, while the SUT is deployed in the environment, and comparing the outcome of the application with expected test results. This procedure includes comparing results between different re-tests (e.g., after regressions). Test execution is recorded in a log along with all information to trace back failures to a certain point of execution. The test results contain passed and failed test instances and those, whose execution involved something unusual or unexpected.

Tests that were not passed can result in an issue which must be handled in the *test incident reporting* activity, which involves analyzing the test results and creating or updating an incident report.

Test Methods, Practices, and Techniques

The extent of testing depends on whether the system artifacts are analyzed with or without execution. The latter case is, called *static testing* and involves manual approaches (e.g., reviews) as well as tool-based approaches (e.g., static program analysis). In contrast, *dynamic testing* executes the SUT with a set of test-cases. Whereas static testing is a method of verification, dynamic test approaches additionally perform validation.

In dynamic testing, software and system engineers perform different practices depending on the types of requirements and level of insight into the SUT. The latter distinction separates testing in three classes: In *white-box testing*, the internal structures of the SUT are exposed and investigated. Test inputs are selected to cover specific paths through the space of inner states of the tested execution logic. In contrast, *black-box testing* relies on a specification, from which testers construct test-cases, and an API for applying the actions in those test-cases against the SUT. Afterwards, they compare outputs with given expectations to gain a verdict about the SUT's correctness. Between those extremes, *grey-box testing* is performed against black-boxes, whereas grey-box test design uses information on system internals.

Especially in black-box testing, expected data must be specified to determine a test execution's correctness. For this purpose, *test oracles* provide a function that maps input data to expected outputs. An oracle may provide different characteristics, which, for instance, includes completeness and accuracy [Hof98]. However, no matter how powerful an oracle is, most real systems are so complex, that only a very restricted, carefully chosen subset of input and output data pairs can be tested. It is subject to an *adequacy criterion*, when a tester is allowed for qualifying an SUT as correct under the given conditions. In test projects, a usual way to systematically select representatives of data to be tested is to divide the domains of relevant input values into equivalence classes and perform a *boundary value analysis*. In this approach, predominantly val-

ues on the edge of the found equivalence classes are selected for the test set because, especially, problematic cases are covered in this way.

Furthermore, the specific quality or type of tested requirement plays an important role. Besides functional correctness (i.e., if the system does what it is made for), non-functional requirements target on manifold qualities. Both types have been documented in ISO/IEC 25010:2011 [ISO16]. According to this widely-adopted standard, non-functional characteristics are reliability, usability, efficiency, maintainability, and portability. Each of them is further structured in sub-qualities, which all can be tested by different techniques. For instance, efficiency is investigated by software performance testing, whereas conformance testing targets on maintainability and portability.

Crosscutting to the mentioned types and approaches of testing, certain actions of this discipline can be automated. Test automation traditionally focuses on how to execute test input and compare test output without human intervention. Whereas manual testing requires long lists of test-cases with human-readable actions, test data, and a lot of repetitive work, automated tests benefit from homogeneous, completely reproducible outcomes. In this manner, many more test-cases can be executed in shorter time spans, because actions are scripted in executable formats, and comparisons of test results are delegated to an algorithm as well. Due to the capability of automatically interacting with an SUT, not only functional testing of large systems but also performance testing benefits from automation.

Additionally, automating test execution supports quickly repeating tests for new versions of the SUT. Such repetitions become necessary due to *regression*, which means that the quality of the SUT potentially decreases with the new versions. This common approach is consequently called *regression testing* and enables test engineers to keep quality over long-term development cycles [Mye79].

However, test execution is only one end of test-automation. Another elaborate step is the creation of test-cases from a specification. The necessary effort can further be lowered by automating the test design, which is discussed in the following section.

2.2.3. Automated Test Design

Managing test data, test-cases, and test configurations can become cumbersome for complex systems. In design and implementation, engineers found methods that help to automate parts of recurring refinement tasks using model-driven software development (MDSD). The recent research adopts MDSD and proposes so-called model-based testing (MBT) as a generative approach that is appropriate for decreasing test design efforts. According to Utting and Legegard, MBT can be defined as follows:

> "Model-based testing is the automation of the design of black-box tests."
>
> [UL07, p. 8]

The authors explicitly distinguish MBT from test approaches that only automate test-case *execution*. Thus, MBT can be understood as an approach to improve the test process efficiency by design automation. In consequence, MBT is especially beneficial if a service accepts a large variety of inputs and calls. This relation is, in turn, a perfect match with the requirements of context-dependence and self-adaptive software service.

MBT can be employed at all levels of testing (cf., Section 2.2.2), whereas it is not compatible with all test techniques, such as white box testing. The latter derives test-cases from design and implementation artifacts (e.g., symbolic execution [Kin76]), whereas MBT only checks the external state of the exposed service. For this purpose, the expected behavior of the system is specified in the form of models, which are later used for automatically generating test-cases. The overall process of MBT is described in the next section.

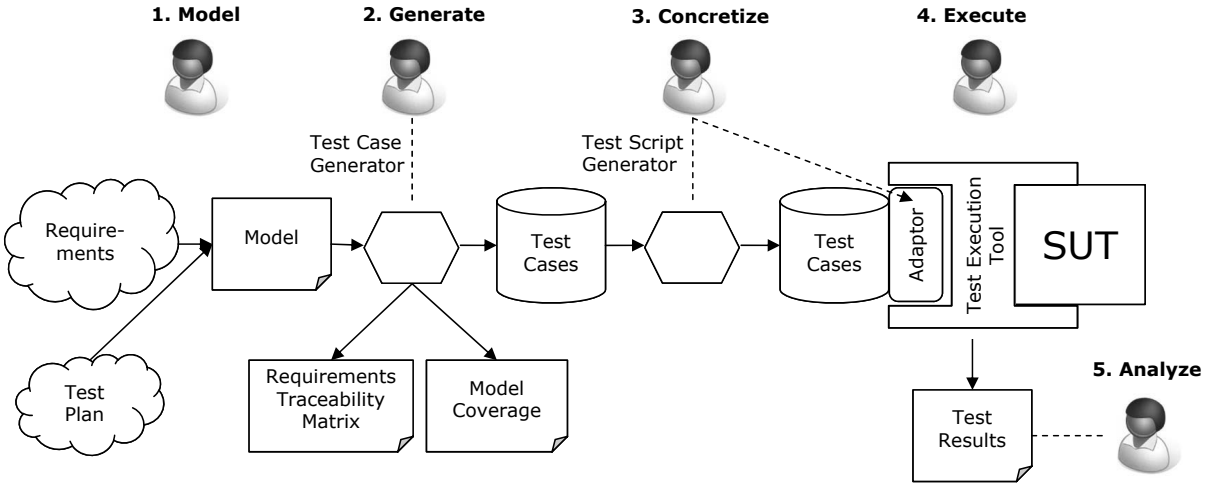


Figure 2.5.: The MBT Process [UL07].

The MBT Process

Automating test design involves certain activities, for which a tool chain must be provided. Figure 2.5 depicts the overall process including necessary tools and artifacts. The process comprises the following steps:

1. **Model:** In the first step, an engineer creates a model. Because MBT is a black-box method, the model has to reflect the requirements of the SUT. Furthermore, the model is more abstract than the real system. For instance, in comparison to MDSD, we can qualify the model as platform-independent if it omits details on technical execution. Additionally, it should be precisely annotated, which model elements refer to which requirements to integrate trace information. The effort and level of granularity that is modeled can be controlled by a formal or informal *test plan*.

2. **Generate:** From the model, a *test case generator* automatically derives test-cases on the same abstraction level as the input model. Each test-case consists of a sequence of operations that can be applied with the SUT. As the majority of systems can run without termination (i.e., that interaction can be executed infinitely), the number of test-cases is infinite, as well. Therefore, the test engineer has to provide a *test selection or test adequacy criterion* that allows for terminating the generation at a specific point.

   Besides test-cases, the generator may produce a *requirements traceability matrix* and *model coverage report*. These artifacts relate test-cases to requirements. After test execution, this mapping can be used to determine violated requirements.

3. **Concretize:** Abstract operations of each test-case have to be mapped to concrete, executable ones. Typically, this step is performed using templates or any other form of transformation mechanism. Alternatively, an adapter can map abstract operations to the SUT's technical interface operations at run-time. Both transformation and adaptation may be realized for different SUT implementations so that the abstract test-cases can be reused.

4. **Execute:** In this step, the generated concrete tests are executed in the SUT. A *test execution tool* schedules and executes all test-cases sequentially. During execution, failures are recorded as well as passed test-cases. The *test results* report this outcome.

5. **Analyze:** Finally, test results can now be interpreted by an analyst. Her/his task is to investigate, which faults caused the failures, and to report them to the developers, who fix them.

The central difference between traditional test processes to MBT is modeling and the generation of test-cases from the models. Without MBT, test-cases are basically derived from an informal or even mental model. With MBT, test engineers benefit from explicit formal definitions that unify the understanding of a system as well as the method how test-cases are created.

Potential Types of Test Models, Metamodels, and Notations

There is a variety of metamodels that are used for generating test-cases. First of all, Utting and Legeard distinguish between four basic approaches [UL07, p. 7]:

1. Generation of **test input from a domain model:** Define value domains of necessary input variables and combine them to generate appropriate test input. As the combination of all input domains may be very large, algorithms, such as *pairwise testing* [CDPP96] can be employed to restrict the considered number of combinations.

2. Generation of **test-cases from an environment model:** Define sequences of environment changes, which are input to the SUT. The generation of sequences can be controlled by a dedicated model, e.g., a statistical one. The advantage is that potentially all states of the SUT can be reached. Still, there are no means for evaluating whether the reaction (i.e., the output) of the system was correct. The only observable reaction is a system crash.

3. Generation of **test-cases with oracles from a behavior model:** In this approach, the model has to include information on the *expected* reaction or output of the system. Thus, the model is taken as an oracle implementation.

4. Generation of **test scripts from abstract tests:** Map implementation-independent test-cases to actual API calls of SUT functionality. This task is identical with step 3 of the MBT process described above.

As it turns out, approach (3) is the most challenging because it demands the largest amount of information; both environment behavior and the SUT's reaction have to be integrated into an appropriate model. However, this approach allows for automating tests to the greatest extent.

Another question is, from which development artifacts the test model is built from [UL07, p. 31f]. The first approach would be to reuse design models from system developers. For testers, this would be beneficial because these models involve a lot of definitions, which are relevant for test design. However, there are several disadvantages. First of all, design models such as structural class diagrams are mostly not detailed enough to describe the *exact* behavior of the system.

Furthermore, the details that describe the system behavior in more fine-granular design models are too implementation-specific so that this approach can be compared to white-box testing. When the code has been generated from the same design models that should be reused in testing, faults manifest in the resulting test-cases, as well. In consequence, taking design artifacts as the only resource in testing is often questionable and, thus, not a satisfactory approach.

Notably, though, a completely new test model would create the maximum independence from design knowledge. The fundamental resource in this approach is a requirements document. Furthermore, it is possible to enrich (e.g., annotate) high-level design artifacts with test-specific information. This approach would create a compromise between complete reuse of design models and the definition of test models from scratch.

A bandwidth of metamodels and notations can be used to represent the input to a test-case generator. Utting and Legeard classify those model types as follows [UL07, p. 62ff.]:

Pre/post (or) state-based notations Each operation is defined together with a *precondition* and a *postcondition*. For instance, the object constraint language[1] (OCL) together with

[1] Object Management Group: Object Constraint Language 2.4, `http://www.omg.org/spec/OCL/2.4`

the unified modeling language[2] (UML) can be used for this purpose. From preconditions, the generator can determine test inputs that are applicable in a certain run-time state and postconditions that are used to decide whether to produce data or if a reached state is correct.

Transition-based notations Test operations are defined as transitions between system states. For instance, *finite state machine*s (FSM) and *UML statechart*s are graph-based models. Each transition can be labeled with concrete actions (e.g., retrieving input or verifying an assertion) that are applied in the presumed state.

History-based notations For instance, in *message-sequence charts* (MSCs) or *UML sequence charts* permitted traces of a system's behavior or time can be defined.

Functional notations Describe a system by a set of algebraic formulas and generate test-cases from these specifications. An example can be found in [Mar95].

Operational notations Define a system as collection of parallel executed processes. Example notations are *Communicating Sequential Processes* (CSPs, [Hoa78]), the *Calculus of communicating systems* (CCS, [Mil80]), and *Petri nets* [Pet62].

Statistical notations If distributions of possible events and input data values have to be defined, statistical models like Markov chains are appropriate. Those model types can be used together with deterministic ones that describe the reaction of the system.

Data-flow notations Instead of control flow, the flow of data through the system is modeled. An example is Lustre [MA00].

To improve the understanding of how test-cases can be generated from a model, in the following an example is provided. Figure 2.6 depicts a finite state machine (FSM), which specifies the expected behavior of a simple web shop. More precisely, the FSM is a Mealy machine, which translates sequences of input symbols to sequences of output symbols. State `Start` is the initial state, which is denoted by the incoming edge from the black pseudo element. Input symbols `AUTH_VALID` and `AUTH_INVALID` abstract from correct or incorrect login data submitted by the user. All other input items symbolize buttons, which have to be pressed to advance to the next shop window. We also abstract from the fact that certain data has to be entered (e.g., while registration), and that it is necessary to select a specific item to be bought because only the SUT's workflow is tested.

To generate test-cases from this automaton, valid sequences of input and output pairs are derived. Because in this specific example, no terminal state was defined, all lengths of test-cases are acceptable. A sample instance is the following:

Example:

`AUTH_INVALID`/*"Login failed"* → `REGISTER`/*"Enter registration"*
→ `OK`/*"Welcome"* → `BUY_ITEM`/*"Added to basket"*
→ `PAY`/*"Please confirm"* → `OK`/*"Please confirm"*
→ `OK`/*"Continue shopping"*

Although the FSM is deterministic, infinitely many of such sequences can be generated. A tester is in charge to decide which of them are more important and which can be ignored. To formalize this decision and steer an automatic generator algorithm, modelers specify selection criteria.

[2]Object Management Group: Unified Modeling Language 2.4.1, `http://www.omg.org/spec/UML/2.4.1`

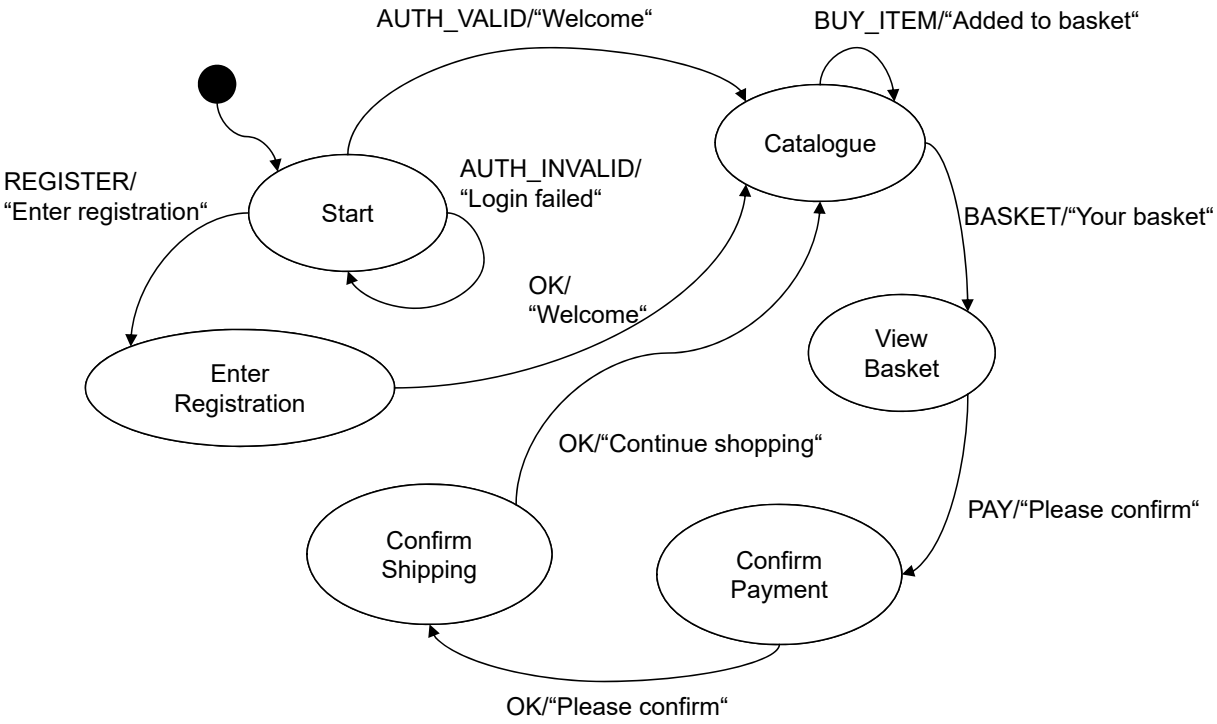


Figure 2.6.: Sample finite state machine for a simple webshop.

Test Selection Criteria

Test suites generated from specifications potentially grow to an enormous size. The combinations of test data domains and paths through behavioral specification creates a so-called *test-case explosion*, which is an exponential or combinatorial growth of the number of potentially executable, valid test-cases. In white-box testing, where code is known to engineers, selection criteria have been found that help to distinguish which parts of the control flow should be examined and which should not. The basic ones are *function coverage*, *statement coverage*, *branch coverage*, and *condition coverage* [Rei05]. Each criterion can be used for two different purposes:

1. Measure how much of a program is covered and determine if this coverage is adequate to the expectations of quality assurance.

2. Determine when to stop testing or when to stop test-case generation.

However, in model-based testing, the appropriateness of a criterion depends on the type of the used model. For the FSM given above (cf. Fig. 2.6) state or transition coverage fit well. Depending on a risk estimation, a tester could decide to reach 50% state coverage. In the first case, test-cases are generated until half of all states of the FSM are reached at least once. For other models, it may be beneficial to define coverage by their specific entities. For textual models, statement coverage is applicable, data flows require criteria based on defs and uses, and object-oriented descriptions work well, for instance, with class attribute coverage.

2.3. Dynamic Variability Management

Despite the discussed techniques for modeling and decision making in SAS, the research field of *software product lines* (SPLs) has developed means to reason on dynamic adaptation, as well. If variation points of a product are modifiable during run-time, the SPL is called a dynamic

software product line (DSPL). Such a notion of variability is a useful foundation for black-box abstraction of SAS and, thus, for a potential test modeling tool.

In this section, background on (D)SPLs is given. Firstly, the idea of SPL engineering (SPLE) is presented. Secondly, the widely-used formalism of feature models is introduced. Finally, the last part of this section presents how the pre-execution means of SPLE and features can be used to represent structure, constraints, and behavior of a dynamic SAS.

2.3.1. Software Product Lines

A central motivation of software engineering methodology is reuse. Divide-and-conquer, abstractions, and inheritance are available in most modern programming languages and metamodels. However, reuse can be lifted up even on the most abstract view of software systems where business strategy or requirements are considered. For instance, when several customers of a software manufacturer demand a product for the same domain but with partly different functional or non-functional expectations than investigated in past projects, it is likely that a subset of the developed artifacts can be reused. Other assets of the system may be changed or even developed newly but should work together with the reusable and shared ones. David L. Parnas even assumed that the commonality in software is predominant in comparison to difference. In consequence, he proposed to refer to such systems as *software families* [Par76]. From this idea, the notions of SPLs and features have been developed.

> "A **software product line** is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment [...] and that are developed from a common set of core assets in a prescribed way." [CN01, pgs. 25f.]
>
> "A **feature** is a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept." [CE00]

Because the definition of a feature is highly related to the interest of a certain stakeholder, it relates to a specific set of requirements, as well. In consequence, an SPL spans concepts between both the *problem space* and the *solution space* of the system [CE00]. Basically, in the problem space, requirements are grouped as features, whereas the solution space contains reusable implementation artifacts.

The process of developing an SPL consists of two subprocesses: In *domain engineering*, the common and variable assets of the resulting SPL are developed. Thereby, a complete software engineering process should be performed including domain requirements engineering, domain design, domain realization and domain testing. In this subprocess, the software manufacturer has to keep track of market requirements by performing sufficient product management. Afterwards, during *application engineering*, the requirements of an individual customer are taken into account, and a specialized application is derived from the existing SPL.

In an SPL, features in problem space and artifacts in solution space have to be related by a mapping. Some of these artifacts may be alternatives, incompatible with each other, or the business strategy prohibits the combination of certain features. To manage such dependencies, various variability metamodels can be used. Foremost, the feature model representation as proposed by Kang et al. is the most widely adopted and extended representation format [KCH+90]. Consequently, Kang's notion of feature models is described in the following and adopted as the fundamental variability model in this thesis.

Feature Models

Kang et al. proposed their feature-oriented domain analysis (FODA) methodology in the early 1990s [KCH+90]. According to the authors, a *feature analysis* has to be performed to capture a

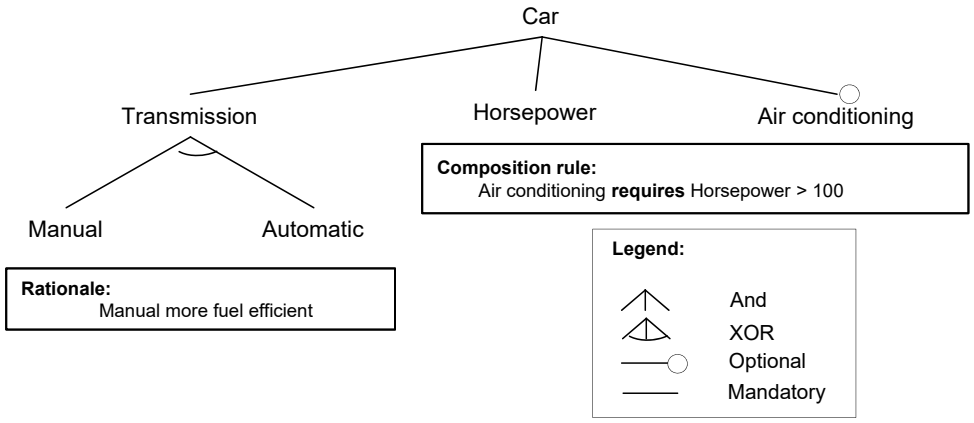


Figure 2.7.: Example of the original feature tree notation (taken from [KCH+90]).

model of the end-user's understanding of the general capabilities of an application in its domain. In FODA, the feature model is represented as a *feature tree*. An example of this notation is shown in Figure 2.7. Each node of the tree is related to a specific attribute of the system that *directly* affects the end-users. The tree structure decomposes the possible attributes of the modeled family of systems along a *consists-of* relation (logical And). In the depicted example, a `Car` may consist of `Transmission, Horsepower`, and `Air conditioning`. Furthermore, `Transmission` can consist of the features `Manual` or `Automatic`. Both latter features are mutually exclusive (XOR), which is denoted by a segment-covering arc between the branches. If a connection ends with an empty circle, the corresponding feature is optional (i.e., it may be part of a valid system).

Kang et al. provide certain additional information to the tree model. The rational denotes that variants with manual transmission are more efficient than those with automatic ones. Based on this information, system instances can be compared to each other quantitatively, or the information can be used for choosing an optimal system. Furthermore, a *composition rule* is defined in the example. It prohibits systems with `Air conditioning` and a `Horsepower` value lower or equal to 100. Such definitions have been called *cross-tree constraints* in later publications.

A feature tree spans a variability space from which several *products* for the modeled domain can be derived, and others are prohibited. For instance, selecting the features

$$\{Car, Transmission, Manual, Horsepower = 100\}$$

forms a valid product, while

$$\{Car, Air\ conditioning\}$$

does not because the mandatory feature `Horsepower` is missing. For automated validation of a product or to generate valid ones, a feature tree can be mapped to expressions of propositional logic, which can be reasoned on by a *satisfiability solver* or *constraint solver* [Bat05]. In this way, it can be automatically decided whether a variant (i.e., a product) is valid w.r.t. the logic relations within a feature model. Alternatively, the solver can be used for automatically deriving valid variants from the formal specification. Thus, in a broader sense, feature models can be understood as graphic notations to express logic and numeric constraints within a variability space.

Variability Realization

Features in the problem space have to be linked to implementation artifacts in the solution space to specify their configuration impact. Preliminary, the implementation has to be variable, so that options can be selected from the related features. In [SVGB05], Svahnberg et al. categorize available realization techniques proposed in the literature. The approach to be preferred

depends on the *binding time*, which may be one of the phases during derivation of the products' architecture, compilation, linking, or at run-time. Another dimension is the granularity of the considered artifact: components, classes, or lines of code. Regarding components, for instance, optional replacements can be specified with the architecture description or as binaries during linking. During run-time, components can be replaced by re-organizing the infrastructure. Having an object-oriented language, class definitions incorporate the possibility of variation within inheritance hierarchies, which can be rearranged within all possible binding times.

On the level of models, for instance, Heidenreich et al. proposed an approach where features within a tree can be linked to arbitrary elements of design models of a target system [HKW08]. This method assumes that the software is developed based on model-driven principles. Feature models and target models are specified under a common metamodel. The mapping is itself a model instance of a mapping metamodel so that the designer stays in the technology space. On the fine-granular level of code, features can be adopted as first-class citizen as done in feature-oriented programming [Pre97], where features are implemented as an extension to object-orientation and integrated by special compiler.

In summary, the concrete binding time of variation point determines the technology space (e.g., a certain programming language or executable format) that has to be used for implementing its activation. Hence, this realization technique has to be compatible to the concrete modeling process of the SPL.

2.3.2. Dynamic Software Product Lines

A product is assembled from an SPL variability space initially before execution time, that is at the time of compiling, linking, or deployment. Different variability mechanisms allow for combining features during these phases [BFG+02], whereas the resulting configuration is static and not thought to be changing at run-time. *Dynamic* software product lines (DSPL) overcome this restriction. The abstractions used in classical SPLE are leveraged to run-time for *reconfiguring* the product. DSPLs are, thus, strongly related to SAS.

In 2006, Hallsteinsen et al. proposed to build adaptive systems based on SPL techniques [HSSF06]. At this point in time, they worked on the MADAM project and investigated that variability modeling could help to describe context and adaptation based on the concepts of SPLs. First, they concentrated on building the adequate architecture to enable the system to adapt and manage all information in appropriately-tailored models. Later on, in 2008, Hallsteinsen et al.'s concepts were developed further resulting in the following definition of DSPLs:

> "DSPLs bind variation points at runtime, initially when software is launched to adapt to the current environment, as well as during operation to adapt to changes in the environment. [...] In DSPLs, monitoring the current situation and controlling the adaptation are thus central tasks." [HHPS08]

Conversely, it is apparent that DSPLs have a very similar intent as systems researched in the SAS community. However, they focus on a specific variability management technique, which is adopted from classical SPLE. In comparison to its static ancestors, the role of market-fit of produced variants loses relevance. Instead, DSPLs adapt themselves according to the environment, respectively context, in the same sense as the MAPE-K loop proposes. Further comparison shows that SAS especially focus on architectures that enable adaptation, whereas DSPL concepts target on reasoning on requirements [ASP13a]. In consequence, both research directions crosscut, but look at problems from different perspectives.

Since that initial proposal, the body of knowledge on DSPLs matured. Bencomo, Hallsteinsen, and de Almeida provide an overview of this development in [BHA12]. The authors present the conceptional model of DSPLs as two interacting MAPE-K loops, whereas one adapts a single product according to environmental changes and the other evolves the whole DSPL. As in (static)

SPLs, problem and solution space are separated and have to be handled by interacting adaptation loops. Besides this architectural concept, it has to be answered how to organize the information on both sides and how to relate them. In the following, appropriate concepts from software research are discussed.

Potential DSPL Models and Mechanisms

In DSPL engineering, additional modeling concepts are required that built on existing SPL ideas but also permit the description of adaptation. Some models that were proposed for this purpose are tailor-made as in [FDB+08]. Fleurey et al. introduce a metamodel, which incorporates concepts for context, variant description, and rules. The context hereby consists of variables with boolean or multi-value domains. Variants of the system map to aspects that can be woven into the system and may be composed by a planner. A rule is triggered by an arbitrary context change and has a condition of the context state as well as an effect specification that activates or deactivates variants. To guarantee consistency, additional constraints, including invariants, define conditions on variants that are allowed for being composed.

However, as the most widely adopted formalism of SPLE, feature models are utilized in the majority of DSPL approaches. Early publications proposed to map features to software components with interfaces that enable dynamic activation and deactivation [TRCPB07]. Over time, researchers found lacks within this understanding and focus shifted towards the question of how the mapping could be enriched. For instance, Cetina et al. combined features and autonomic computing [CGFP09]. Their feature model is mapped to elements of a model at run-time (MRT, [BBF09][ABCF11]) and describes devices, services, and channels of a smart home system. The mapping operator is called *superimposition* and directly relates a feature to a set of entities within the MRT. The authors also claim that their architecture comprises algorithms able to create reconfiguration plans based on the selected features and mapped solution space properties.

In contrast, several approaches propose to mark certain features as dynamic so that they can be distinguished from static variability [DMFM10]. Such distinction leads to different fixed binding times of features. For instance, Lee and Kang bundle sets of features in *binding units*, which have certain binding times [LK06]. Features to be put into a single unit are determined by their dependencies within the feature tree. A deeper analysis how to relate features to certain binding times reached Rosenmüller et al. in [RSPA11]. They define a comprehensive strategy how feature binding units can be derived from a feature model and mapped to different binding times.

In [ACF+09] and [ACL+11], Acher et al. propose another DSPL system, which again uses feature models as basic formalism. Both context variability and system variability are described in this format. They form two initially independent SPLs. Whereas each context variant (i.e., a valid feature configuration derived from the respective feature model) describes a potential environment state, each system variant represents a potential adaptation mode. In this sense, a context change corresponds to the reconfiguration of the context SPL, i.e., the selection of another context variant. In consequence of such change, a new system variant has to be configured. To specify, which system variant is appropriate for a certain context state, constraint-based rules are defined by the designer. A rule consists of a left-hand side (LHS), which selects context variants and a right-hand side (RHS), where the respective system variants are selected. Both sides are specified in propositional logic in the same manner as in composition rules in FODA (cf. Section 2.3.1). Each rule is an expression of the form *LHS implies RHS*. Hence, the context can be monitored for context states where a LHS is matched. If so, the RHS evaluated, and the new system variant is deployed. In this way, the complete causal connection between context change and adaptation decision can be modeled with feature and constraint context within a common language family.

However, constraint-based rules inherit several problems. Firstly, they have to be prioritized

as multiple of them could be triggered by a single context change and may have conflicting consequences concerning the adaptation result. Secondly, there is no state that permits to execute the rules in arbitrary situations. This lack of expressiveness misses the fact that certain configuration decisions are potentially only accessible in a specific operation phase. Thirdly, no technique for solution space mappings has been proposed so that they have to be specified externally. To overcome all these problems, Helvensteijn introduced a Mealy-machine-based formalism to organize reconfiguration of DSPLs [Hel12a]. The concept is named delta modeling as both reconfiguration and change are defined in the form of incremental delta operators. In delta modeling, the separation between context and system variability space is based on different notions. Again, a feature model defines valid and invalid combinations of features that are allowed for setting up a specific variant (i.e., product). Rules associate the conditions in the form of modifications on the active feature selection with reconfiguration operations in the solution space. Within the Mealy machine, each state denotes a valid configuration and transitions are guarded by the given rules. In this way, the DSPL becomes stateful, and no precedence of rules is required anymore.

In conclusion, the body of knowledge of DSPL engineering crosscuts with the one of SAS. Several architectural solutions and approaches that organize the variability within the operational space of DSPLs are useful additions to SAS principles as the MAPE-K loop.

3. Related Work: Existing Research on Testing Self-Adaptive Systems

Testing, as briefly introduced, is the most-adopted method of quality assurance. The systems engineering discipline put much research effort in the question "How to make systems dependent?" and has produced a large body of knowledge in this domain. However, like in other domains of engineering, SAS research predominantly develops design concepts first. In consequence, literature covering quality assurance, and testing, in particular, is underrepresented.

Two of the first researchers, who aimed at filling this gap, are da Silva and de Lemos. They proposed to use workflows for generating integration test plans for SAS [dSdL11]. In their system, instances of an SAS are assembled from components at run-time. Each time a new component has to be integrated into the already running configuration, the SAS executes a set of integration test-cases. The latter are bundled with the component to be deployed. The authors focus on the construction of workflows that represent test plans, which involves the definition of the correct order of tests to be executed and the generation of stubs. Despite this approach uses test-cases to determine correct integration, it can be characterized as a technique of fault tolerance because it avoids the integration of conflicting components and, thus, keeps the system consistent. There is no overall test plan, which contains information on components to be tested in *predefined configurations*. Consequently, this approach lacks reproducibility and applicability as a technique of quality assurance before delivery.

As in da Silva's method of at-run-time testing, many proposals avoid the challenge of SAS test complexity. However, despite the rareness of actual pre-delivery test approaches in this body of knowledge, SAS engineering is not an isolated domain so that several related domains can contribute concepts to be reused for the SAS test problem. On the one hand, context-aware applications are an ancestors of SAS, and some problems that respective engineers were faced are transferable. On the other hand, the domain of DSPLE (cf. Section 2.3.2) was initially only loosely coupled with SAS research so that some approaches were discussed out of the sight of the SAS research community. In summary, we identify three different research domains to be considered as relevant:

1. **Context-aware applications:** A research domain that developed already during the 1990s and aimed at constructing mobile applications that react to changes in the device's location, connectivity, and the profile of the user in control.

2. **Self-adaptive software:** This domain developed in the previous decade (2000-2010). Testing was only considered rarely until now in this area.

3. **Dynamic software product lines (DSPLs):** Initially, SPL research only considered how an implementation could be configured for different customers based on individual

and shared requirements as well as related features. Lifting this problem to run-time, required to advance the found principles to a more dynamic view and resulted in DSPLs. Thus, DSPLs can be seen as a special form of SAS, where the models and implementation means from SPL research are adopted.

In the following, several projects and publications from the named research domains are discussed in detail.

3.1. Testing Context-Aware Applications

Context awareness means that an application can alter its behavior not only in reaction to explicit inputs but, additionally, to its situation in a certain physical or computation-related environment. Initially, this approach was especially seen as means for better adhering to a user's needs [LS00]. In contrast to SAS, the principle of context involvement is not conceptualized to the degree as, for instance, control loops in SAS are, and, thus, the focus lies on how to integrate context into applications.

Early, it was recognized that due to the involvement of different context sources, verifying an adaptive application requires additional effort. Thus, the simulation of context events was of special interest. One of the first approaches to this challenge has been proposed by Broens et al. [BH06]. The authors developed SimuContext, a simulation engine that mocks context data based on an explicit context specification. In this way, real context sources can be replaced by simulated ones so that the effects of enforced context change can be observed in the SUT. SimuContext comes with an explicit model of events that can be extended by developers so that the framework can be employed in different application domains. In principle, SimuContext is a straight-forward approach for testing context-aware applications because it adopts the idea of providing mocks for inputs from traditional testing. Based on this concept, contextual test data can be exactly specified and variants of it generated. Also, the infrastructure allows for enforcing this data as input of SUT so that the expected behavior can be verified.

A comprehensive project on SAS and context awareness is MUSIC. It contributes design methods, a system architecture, a middleware platform, and a simulation environment for context-adaptive systems, including respective tooling and prototypes.

In MUSIC, adaptation incorporates the interchange of components and change of parameterization. All possible adaptations are specified in composition plans, whereas an optimization component selects variants based on utility functions. A utility function gives a measure how beneficial an adaption is in a certain context situation. Context data is contributed by context clients to a context manager. The data flows in as an event stream, which is organized by the MUSIC middleware and interpreted continuously.

To verify expected behavior, MUSIC also incorporates a context simulation tool, which intercepts the event flow [KPPR10]. Simulation scripts produce invocations on services and context events in a timely order and trigger adaptations, which can be monitored for verification purposes.

Context-aware applications are expected to collect data from their environment and derive a context model, which is used to adapt services according to the recognized situation. In the technical realization, gathering context data for subsequent reasoning requires the integration of sensors, abstraction, a common data model, and appropriate reasoning mechanisms. To unify and ease the development of context-aware systems, several *context middleware* frameworks have been implemented. Components of the middleware provide a common interface to explicit or implicit (i.e., computed) contextual information. Thus, context-aware applications can be developed by using this interface and without the need to deal with the heterogeneous hardware devices of the sensor equipment.

Discussion. Context-aware applications are a predecessor of SAS. Consequently, there exist characteristical requirements of testing context awareness, which are also relevant to SAS testing:

- **Enforcing** context: to verify a system against requirements and to find failures, the test should be able to change context willingly to a different state, under which the SUT should to be tested. This ability allows for step-wise investigation of how the system behaves under different conditions. In SimuContext, conditions are enforced based on a specification, whereas MUSIC provides this functionality based on scripts.

- **Generative** creation of context variants: In contrast to MUSIC's simulation scripts, the authors of SimuContext proposed to automatically derive variants from specifications. With this approach, the authors target on avoiding manual and, thus, very costly definition of a multitude of context situations. Instead, they prefer automated generation from declarative formats, which is a reasonable method to avoid dealing with adaptation-related complexity.

- **Stateful** context change: MUSIC simulation scripts introduce an explicit order, in which the context events occur. For systems that react to changes it can be enormously important if, for instance, a context value increases or decreases over time. Thus, one requirement for SAS testing, it the ability to enforce not only situations but also their order of occurrence.

Parts of these requirements were already claimed by Wang et al. in [WER07]. According to them, it has to be identified *where* and *which* context situations interfere with the actual system behavior. Furthermore, they recognize that the variety of possible interactions between different context values and the system behavior can be enormously large and has to be dealt with in test generation.

3.2. The SimSOTA Project

Within the SAS research field, the state of the affairs (SOTA) project [ABZ12] by Abeywickrama et al. offers a more comprehensive simulation and validation toolset, named SimSOTA [AHZ13]. The approach's central paradigm is to model feedback loops explicitly as first-class entities. Thus, different patterns of self-adaptive control flows can be constructed.

This fundamental assumption of SOTA is that all adaptable parameters of a system can be described as property dimensions of a value space. These properties constitute an n-dimensional state space, in which the system's execution navigates. A system is *self-aware* if it can autonomously locate itself and track its movement inside this state space. Furthermore, the system is *self-adaptive* if it can dynamically direct its trajectory through the state space. Hence, the direction capability can be implemented using feedback loops.

The constraints that define under which conditions the system is expected to operate are modeled as n-dimensional regions within the state space. Similarly, goals define a region, to which the system aims at to direct its trajectory through state space. New goals are activated as soon as a certain specified region is entered. Each goal is considered as post-condition, which the system approximates to by adaptation. During this process, the software adheres to another class of constraints, the so-called *utilities*, which provide heuristics that guide the navigation.

Feedback loops are in charge to control the state space trajectory by deducing parameter changes from monitored sensor data and apply manipulations using effectors. In SOTA, feedback loops do not have to be monolithic. Instead, multiple cooperating loops can be modeled. Each feedback loop manages at least one *service component* according to an associated goal set. Additionally, several service components may share a set of common goals and, thus, form a *service component ensemble*.

For the interaction of feedback loops, either inter-loop or intra-loop coordination can be applied. For inter-loop, multiple sub-loops implement the various phases of MAPE-K. For intra-loop coordination, different mechanisms can be applied by autonomic managers: (1) *hierarchy* means an outer loop is directly controlling an inner loop, (2) *stigmergy* where a shared subsystem is manipulated by multiple loops, and (3) *direct interaction*, which incorporates mangers communicating peer-wise; typically in to control a common resource.

The SOTA model is well-suited for constructing arbitrary MAPE-K systems and for describing their feedback-driven behavior. To allow engineers for modeling and simulate SAS based on SOTA, the SimSOTA environment has been developed. It provides capabilities for modeling, execution, and debugging-like validation of SAS. Here, modeling founds on UML 2.2 so that feedback loops are represented by activity, sequence, and composite diagrams. The single phases of MAPE-K are encapsulated as packages within a common super package. Using the synchronization mechanisms, the interaction between multiple feedback loops can be modeled. Each phase may save information to or query information from a knowledge model.

It is possible to attach breakpoints to the model so that its execution can be debugged. The user can execute the model stepwise, suspend the execution, or resume it. During this process, the execution state is visualized and, thus, can be inspected in detail. Depending on how formal the model is defined or which information is available at run-time, the user may provide data by using a run-time prompting feature (i.e., a dialog). With the help of this tooling, SimSOTA models and simulates a complete SAS and allows for observing and debugging so that errors can be diagnosed and traced to the original fault.

Discussion. According to the authors, SimSOTA provides a general-purpose simulation platform for SAS, which can be instantiated application-independently. Furthermore, the approach can be stated quite powerful due to the comprehensive tool support for modeling, simulation, and validation.

The SimSOTA modeling environment enables to observe how a detailed model reacts to context changes in the loop. However, in contrast to the features of test approaches for context-aware systems (cf. Section 3.1), it lacks expressiveness for means for defining different variants of context and enforcing situations in a specific order or generate variants from a model.

In contrast, SimSOTA copes with different characteristic requirements of testing self-adaptation:

- **Parametric** adaption: The test specification, which is input to the simulator, allows for defining how parameter values in the SimSOTA state space change in reaction to context change.
- **Behavioral** adaptation: By explicitly modeling control loops with the expressiveness of UML, the behavioral flow of the simulated system can also be defined in dependency to observed context flow.
- **Stateful** adaption: Due to the composition of multiple control loops, expected adaption can be made dependent on its history. Consequently, an adaption to the equal context flows must not be equal if there are different histories, which can be verified.

Abeywickrama et al.'s work [AHZ13] claims that explicitly modeling feedback loops is beneficial for designing and analyzing MAPE-K-based SAS. Hence, a monolithic model is not desirable because the system can incorporate a significant number of and context property dimensions, for which an SAS adapts, can be manifold. For this reason, single components only access partial information so that their models can be decoupled. Additionally, the behavior of SAS incorporates a set of complex uncertainties originating in events where certain environment conditions cannot be anticipated in their function or their timing.

Unless SimSOTA provides a debugging-based validation method for SAS, it cannot replace testing. The major drawback is that environment situations that have to be validated cannot

be systematically enforced as there is no description method for such a requirement, which also hinders reproducibility.

3.3. Dynamic Variability in Complex Adaptive Systems (DiVA)

The DiVA project aimed at providing design methods and a technical framework for building and running self-adaptive software. Therefore, several variants of the system can be defined using a variability model. The variation points within this model are related to aspects that can be woven together to configure the system. Decision-making reasons on quality of service (QoS) properties of the system's context and goals, which have to be explicitly modeled by the user.

In the DiVA project, the authors also target on finding appropriate means to validate the constructed SAS applications. The results of this research were mainly published in the project deliverable D4.3 [MBS10]. DiVA's validation approach is separated into two phases. The first one is called *early* validation. This phase utilizes design space artifacts, which are subject to a simulation. Such artifacts comprise adaptation rules, a context model, and several variability models.

The second phase is an *operational* validation. Hereby, the system is executed and examined at run-time against adaptation rules and specific context instances. Details of both validation parts are described in the following.

Early Validation Only design models are used for validating the software so that the engineer can discover faults without actually executing the system. The employed models comprise the adaptation logic, structure, and the context model of the validated application. From the adaptation logic, adequate environmental data is deduced. The validation is then performed by *simulating* the system under such scenarios of environment change. During the simulation, the adaptation logic is used to compute a set of system configurations according to the current environment situation. The correctness of a configuration can be determined by providing a manual oracle.

The example, which is outlined in [MBS10], encompasses a robot application that aims at exploring and mapping a given room. For this purpose, an agent application, which is running on the robot, delivers the gathered information from several sensors to a server. The self-adaptive software is in charge to reconfigure the system according to relevant context input. Both the context's and system's variability are represented as *feature tree* (cf. Section 2.3.1) plus respective constraints on the features in this tree. In DiVA, there are additional adaptation constraints, which establish conditional relations between multiple feature trees. Thus, it can be formally defined that specific system variants require certain states of the monitored context. In this way, it is possible to define that, for instance, that the robot software can only activate its Bluetooth feature when a Bluetooth signal is available in the context.

Variability models limit the set of allowed combinations between context states and system configurations. However, those models are not sufficient to express which system configuration will be chosen by the adaptation mechanism. Instead, this decision must be implemented by a fixed set of rules or, alternatively, by an optimization algorithm that selects the *best* solution from the set of valid ones. For design-time simulation, the DiVA authors propose to determine the correctness of such decisions by using a manual (i.e., human) oracle.

Another issue is the combinatorial explosion of possible context variants that have to be tested. As soon as the model exceeds a certain limit, the number of valid configurations become unmanageable, which is a well-known problem in testing, as it always occurs when multiple input variables with huge domains have to be combined. The problem then constitutes typically in the form of test-case explosion. To avoid this problem, so-called adequacy criteria are used, which define an exact measure that determines when a test-case generation should be canceled (cf. Section 2.2.2). DiVA proposes several of adequacy criteria, such as simple, pair-wise,

and dependency-based coverage over selected contextual features.

Operational Validation In contrast to early validation, the operational phase also takes run-time information into account. Thus, faults in the system's implementation and especially the decision maker's implementation should be uncovered. For this purpose, again context variants are provided to the system to validate the resulting adaptation decision. As the implementations of the run-time employed reasoners are not modeled, they appear as black-boxes to the testers. Furthermore, inside a black box, a certain state can be established that impacts decisions differently after each request. In consequence, a central point of operational validation in DiVA is an additional temporal test dimension.

In the operational validation phase, coverage is approached by so-called *multi-dimensional coverage arrays* (MDCA). The latter are sequences of configurations, where each step sets a certain variant for a variability dimension within the SUT's context. In this way, also a temporal order of variants is modeled. MDCAs are generated automatically to minimize the number of test-cases when many feature interactions have to be considered.

Discussion. DiVA manages variability in context and system configuration by feature models and constraints. Context variants are modeled explicitly, whereas the order of the application is created in a generative process. However, the DiVA publications mainly focus on special coverage criteria for SAS so that a direct comparison to the before discussed approached is ineffective . In consequence, the challenges of defining expected adaptation, no matter if it should be parametric, behavioral, or stateful, are not tackled. Despite this gap, DiVA recognizes the complexity of SAS state spaces as a serious problem in testing:

> "The exponentially growing size of the context space stresses the need to select a limited, but representative number of environments, and test the behavior of an adaptive system against them. This may allow developers for ensuring the correctness of the decision-making process and prevent future incidents. "[MBS10, p. 57].

To overcome this problem, it is necessary that the concepts and tools provided by a specific approach incorporate model-inherent means for **limiting state space**. In DiVA, this task is approached by, firstly, adaptation constraints between context and system variability space and, secondly, by specially designed adequacy criteria.

3.4. Other Early-Stage Research

Despite the discussed, already finished, projects, several research groups published statements on ongoing work in the field of SAS testing. In 2013, Nehring and Liggesmeyer outlined a test framework on component level [NL13]. In their concept, they focus on transactional changes in the structure of software components. For this purpose, they perform six verification steps: In step (1), a system model is derived, and a representative workload has to be prepared by developers or system engineers. Step (2) is to verify if the structural reconfiguration is performed correctly. This task is performed by a system architect, who is in charge to check the correct order of configuration actions, the validity of the reconfiguration end state and the affected quality of service properties. In step (3), data integrity is checked while increasing load stresses the system. Step (4) considers that components that are replaced during reconfiguration have to correctly transfer their state to the substituting component. In step (5), the interaction between system transactions and the adaptation itself is checked for failures. Finally, in step (6) , the non-substituted components are examined for identity. For the complete workflow, the detailed technical and conceptional descriptions are still missing so that it is not possible to compare it directly to the previous approaches. However, throughout the work of Nehring and Liggesmeyer,

several requirements for grey-box level tests are recognized. Especially, identity, state transfer, and the order of reconfiguration are named.

Another on-going work has been published by Eberhardinger et al. in [ESKR14]. They aim at testing a self-organizing system, whose behavior is so complex that it cannot be explicitly specified at design-time. The test concept is based on a *Corridor Enforcing Infrastructure* (CEI), which monitors and verifies predefined constraints on system properties during testing. Such a corridor is called *Corridor of Correct Behavior* (CCB) and can be generated semi-automatically or even completely automatically from requirement documents. To stress the SUT, environment data can also be generated from requirements and state machines. The result is a set of *Environment Variation Scenarios* (EVS). Eberhardinger et al. propose to run these EVS for single agents of the composed SAS, whereas scenarios for interaction agents should be generated from communication protocols. The approach, thus, covers all stages of software testing, that is unit, integration, and system level.

Finally, Wang et al. utilize code fragments in context-aware applications that access the middleware's interface [WER07]. From these so-called context-aware program points (CAPPS), test-cases can be generated. The prerequisite of using a middleware states a general assumption. However, the test generation concepts are independent of a specific middleware implementation.

Prerequisites to the generation processes is a context-aware program P and an initial test-suite T comprising test-cases $t \in T$. The approach aims at enhancing T by inserting context manipulations. A component named `CAPPS Identifier` takes P as input as well as a set of signatures that define API methods of the employed context middleware. The authors' example middleware is designed in form of an observer-subscriber architecture. The context-aware program registers at the middleware and is notified about context changes. An application may also be registered with multiple handling threads so that the middleware potentially provides shared data objects between these threads. Context data is then delivered through the CAPPS (i.e., the statements where the API signatures are accessed) and is expected to be further used by the application. The CAPPs themselves are statements of one of two different types: statements dependent on reading or writing on 1) context data objects, and 2) inferred objects shared between handler threads. CAPPs of type 1) are found by utilizing side-effect analysis [LLH05], whereas type 2) is found using escape analysis [RH04].

During analysis, the `CAPPs Identifier` produces control flow graphs (CFG) for all handler threads. Each of the parallel executed CFGs includes several nodes, which directly relate to CAPPs. As the CAPPs are context-dependent, the CFG can branch to reflect different behavioral consequences of a context evaluation.

The CFGs are input to the `Context Driver Generator`, which traverses them and produces a set of *drivers* D. A driver is a sequence of nodes of operations that drive the later test execution. Due to the parallel execution of context handlers, drivers that run across multiple threads are in particular important to be covered.

In the next step, the `Program Instrumentor` adds certain method calls to the code of program P to encapsulate the handling of each CAPP. These method calls notify a scheduler component (that is included in the code base) on the entry and exit of a CAPP execution with parameters on the current thread, CAPP identifier, and a driver identification. The entry call waits until the currently executed driver reaches the respective CAPP so that both driver and the tested program run synchronously. A CAPP is marked as successfully executed when the exit method of the scheduler is called. The resulting instrumented version of program P is called P'.

The last required component is the `Context Manipulator`. It runs test-cases $t \in T$ against the program P'. The execution of a test-case runs in parallel to the execution of a context driver so that different test outcomes may be generated. As soon as a test-case directs the program to an instrumented code location, the driver is awaited to manipulate the context situation. Thus, it is possible to drive each test-case through an adequate set of CAPPs (according to a predefined *context adequacy criterion* on CAPPs). In this process, the control flow may as well be directed

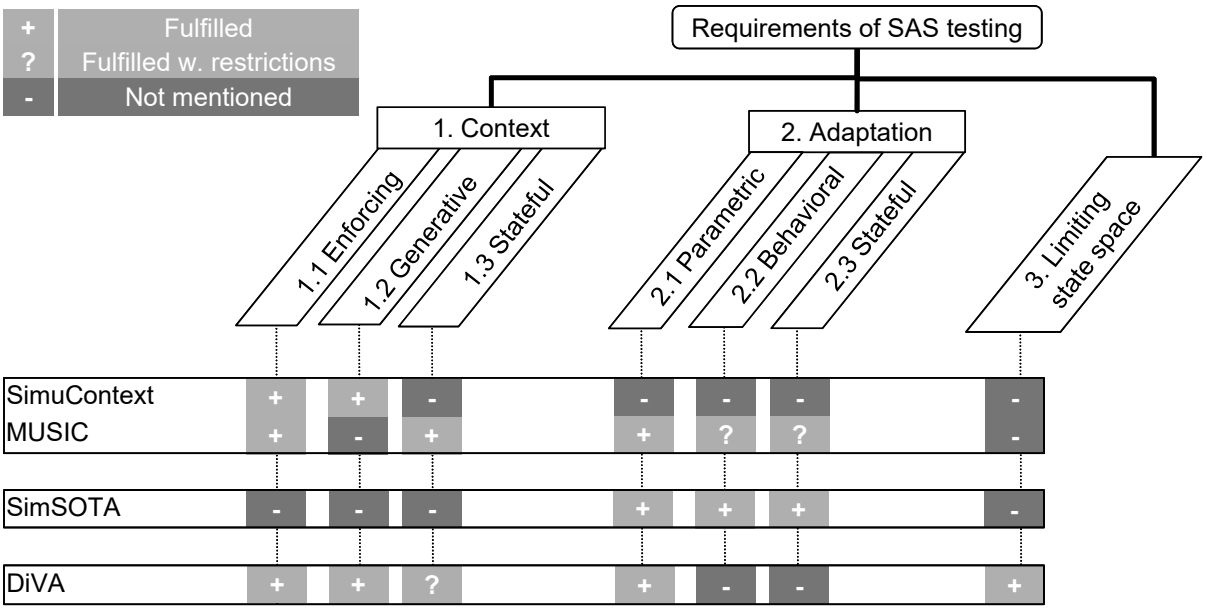


Figure 3.1.: Taxonomy of requirements of model-based SAS testing.

to points where certain CAPPs cannot be reached anymore. In this situation, feedback is handed out to the driver generator, which then generates re-scheduled, more appropriate drivers. The result of the process is a measure of the reached coverage and test-cases that have been enriched with interleaved context manipulations.

However, although Wang et al.'s approach produces a test-suite, the user is required to run the system for deriving test-cases so that only real behavior is detected leaving out behavior that is expected according to a specification. Consequently, the method is adequate for regression testing but not in general for black-box testing.

3.5. Taxonomy of Requirements of Model-based SAS Testing

Due to lack of attention for testing in SAS research, also a general categorization of test approaches is still missing. However, in the previous chapter, several characteristic requirements appeared. Figure 3.1 organizes these requirements in a classification tree. We distinguish between three general taxonomic categories of requirements, which are context testing, adaptation testing, and limiting the state space of the test.

All approaches, which have been discussed in detail in this chapter, are put in relation to the extracted requirements. Those projects, which stem from context-aware computing, provide mostly features on the context side. SimuContext enforces situations, which are generated from a specification. MUSIC uses manually written scripts, which do the enforcement job by stating concrete commands of context change and could, with some effort, even create a stateful coverage of context. On the level of modeling expected adaptions, the authors of SimuContext do not provide information, whereas MUSIC test scripts can check for certain properties and—with restrictions—also behavioral and stateful adaptation. However, due to the missing modeling capabilities of scripting, the effectiveness of this concept is questionable. Finally, none of the named approaches provides any means for limiting the state space to be tested.

SimSOTA's character is quite different to context-aware testing. The approach does not propose how to generate contexts, but models expected adaptation in detail. It provides methods to change parameters of the SUT and to check behavioral changes by modeling feedback loops, which also establish a state space of adaptation. Despite this feature richness, functionality for limiting state space is again missing.

Apparently, DiVA provides the most advanced techniques for SAS testing because parts of the state space of context and adaptation can be modeled together. A drawback is that the stateful construction of context is driven by heuristics so that the user does not have a direct influence on the generated sequences. Also, DiVA has no explicit model of adaptive behavior nor state of adaptation. The focus of DiVA obviously was on coverage and limiting state space, for which, consequently, it performs best.

Each discussed test approach has several pros and cons. Because this thesis aims at finding a generic approach and framework for model-based testing SAS, it has to completely implement the, possibly generalized, capabilities as the existing approaches. In the following part of this thesis, the named challenges are tackled in a conceptional manner and, later, by providing a test environment, which realizes those concepts in a reference architecture and implementation.

Part II.

Methods

4. Model-driven SAS Testing

This chapter is based on the following publications:

- Georg Püschel, Ronny Seiger, Thomas Schlegel (2012). *Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets.* In Proceedings of Modiquitous workshop, pp. 1–4, 2012.
- Georg Püschel, *Testmodellierung für mobile Anwendungen.* In Proceedings of Innovationsforum Open4Innovation, pp. 65âĂŞ69, 2012.
- Georg Püschel, *Test Modeling of Dynamic Variable Systems Using Feature Petri Nets.* Technische Universität Dresden, Fakultät Informatik. ISSN 1430-211X, TUD-Fl13-01-Sept. 2013. Technical Report, 2013.
- Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Uwe Aßmann, *A Black Box Validation Strategy for Self-Adaptive Systems.* In Proceedings of ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, pp. 111-âĂŞ116, 2014.
- Georg Püschel, Sebastian Götz, Claas Wilke, Christian Piechnick, Uwe Aßmann. *Testing Self-Adaptive Software: Requirement Analysis and Solution Scheme.* International Journal on Advances in Software, ISSN 1942-2628, 7(1&2), pp. 88âĂŞ-100, 2014.
- Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Thomas Schlegel, Uwe Aßmann, *A Combined Simulation and Test Case Generation Strategy for Self-Adaptive Systems.* Journal on Advances in Software, 7(3&4), pp. 686-âĂŞ696, 2014.

As introduced in the previous chapters, enabling testers to find failures in SAS is a complex task. In related work, we have shown both that there are open problems that not have been sufficiently solved by state-of-the-art research and that none of the existing solutions covers the complete set of requirements. This thesis' goal is to overcome this research gap.

4.1. Problem/Solution Fit

The major theme in SAS testing is the engineer's need for a mature approach that supports her/him to lower effort and to keep the overview of the complex conditions within the validated behavioral space. Consequently, the proposed solution shall be built on *automation* of tasks that have to be performed to design and execute SAS tests.

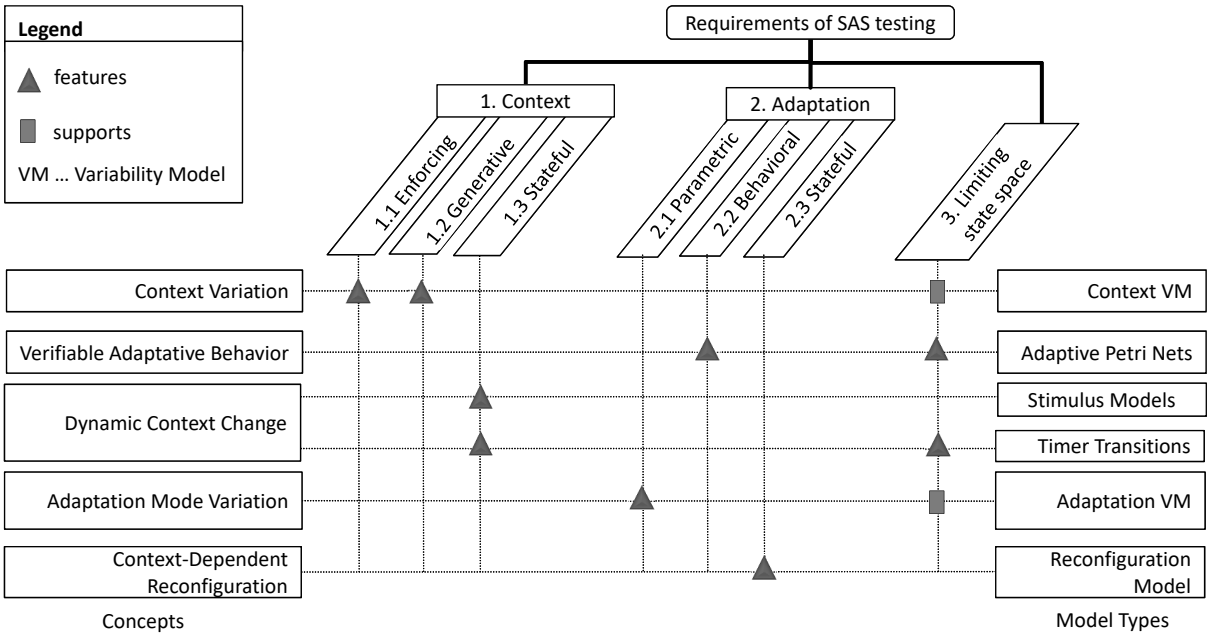


Figure 4.1.: Problem/Solution fit. *The upper of part of the figure shows the previously found requirements of SAS testing, whereas the left side lists the proposed solution concepts. Both requirements and solutions are related by an entry denoting that the concept directly features the requirement or, otherwise, supports its solution. Each concept is implemented by one or more model types, which are denoted on the right side.*

As discussed in previous chapters, test approaches for SAS have several requirements, which are not yet sufficiently met by existing works. To fill this gap, this chapter shall propose a solution, which is fostered by concepts that stem from the traditions of model-based testing, simulation, and variability management. Each concept features one or more requirements, which have been elaborated in Section 3.5. The proposed solution suggests a set of concepts and corresponding model types, which facilitate the required expressiveness for each concept. With this background, the overall solution is flexible in a sense that a model type can be replaced with another one as long their requirements for expressiveness are still met.

Figure 4.1 shows the relation between requirements, concepts, and model types. A concept can also support state space limitation without be directly designed for this purposes. The model type, which is proposed to be used along with a concept, is denoted on the right side of the illustration.

The first concept copes with the necessity to enforce different variants of context. To automatically generate different contextual situations from a formal specification, variability modeling shall be applied in this thesis. The respective model type, which we put hereby into place, shall be named *context variability model* (context VM). It specifies the classes and decision points, which can be used to generate context situations. Due to its built-in abstraction capabilities, it also enables for classification of contextual situations.

Adaptation not only incorporates the change of system properties and composition of its components, but also the change of service behavior that a system provides. From a black-box perspective, the protocol of such services is relevant. *Adaptive Petri nets* cope with the challenge of modeling adaptive behavior. This model type extends classic Petri nets using constraints about the recent variant at run-time.

Change in contextual conditions is the central motivation behind self-adaptivity. Whether the SASuT behaves correctly can only be decided, if it is examined under an adequate set of dynamic change scenarios. For this purpose, the proposal equips testers with *stimulus models*, which describe how contextual dynamics can be established. Stimulus models are fundamental

to the definition of stateful dynamic contexts.

As done for contexts, we can use variability modeling and equivalence class modeling for organizing potential system states that refer to adaptation. In the following, such states are called *modes* and the respective representation is called *adaptation VM*. Based on such a model, also parametric adaption can be specified efficiently.

Between context change and dynamic adaptation, a causal connection has to be established. In this thesis, such a connection is created by the use of symptom events, which are produced within stimulus models and consumed in *reconfiguration models*. The latter two define the explicit adaptation behavior and, thus, allow for modeling stateful adaptation. The problem of defining which contexts should be enforced in which adaptation modes is tackled by *timer transitions*. These are an extension to Petri nets and enable them to take control of the context variation at certain points in their execution. For this purpose, timer transitions define the production of certain portions of discrete time controlling stimulus models. Thus, context changes are controlled in a quantitative manner, and a reverse control flow is established, which also features the limitation of state space.

This chapter contributes the following proposals coping with the SAS test challenges:

- Model-driven methods for testing before non-covered regions of an SAS' state space based on generated test-cases or simulation in the loop
- A comprehensive formalization in the form of metamodels that implement all aspects of the SAS test methods
- Timer transitions as novel concept for communicating between models in a quantitative manner
- A conceptional infrastructure to foster the dual use of test-case generation and simulation in the loop
- A step-wise extension starting from Petri nets, which allows for employing above concepts and customizing them for the required level of coverage

The chapter starts with an example application for drone-based surveillance in Section 4.2. The example shall be modeled and extended in the following conceptual sections to illustrate the used models and their interactions. Subsequently, in Section 4.3, basic concepts, including Petri nets as a formalism for behavioral modeling, are introduced, followed by a discussion of the complete list of concepts in the order presented in Figure 4.1. In Section 4.4, different adequacy criteria for test selection are discussed, which can be used with the proposed model stack. Furthermore, in Section 4.5, the adequacy of the employed models is discussed. Finally, Sections 4.6 and 4.7 demarcate the approach from related work, summarize, and elaborate further side issues.

4.2. Example: Surveillance Drone

For the illustration of the proposed concepts and models, we introduce an example of an autonomic system. The system consists of an unmanned air drone, equipped with a camera, a height sensor, and a satellite-based navigation system.

The drone is intended to fly, starting from a home position, along a predefined path; spot for remarkable situations and record them. Depending on the battery state, the light and weather conditions, the inferred level of threats, or other parameters, a control loop may decide to leave the planned track or even fly back to the drone's home station. The control loop algorithm is executed on-board, which allows the drone for adapting itself and operate autonomously.

The range of applications of such an autonomic and self-adaptive system is manifold: For instance, catastrophe areas can be continuously inspected [MCMO10] and lost people searched

for [GGC+16]. With the goal of complete automation in execution and—later on—testing, the example shall incorporate no human interaction so that it is possible to automate test execution completely. Moreover, it is assumed that a battery loading procedure is performed without manual intervention. Precise models for this example SAS are step-by-step extended in the following sections.

4.3. Concepts and Models for Testing Self-Adaptive Systems

Only concepts that integrate with ease assemble an effective overall solution to the general problem of testing SAS. The consistency of the proposed models shall hereby be established by formalizations of the exact syntax and semantics of all solution artifacts. Moreover, this section shall explain why the single concepts leave gaps that each is dealt with in the follow-up section.

4.3.1. Test Case Generation vs. Simulation in the Loop

Generating test-cases from models in traditional MBT is a systematic process, which leads to repeatable regression tests and reproducible results. However, in some setups, the SUT's context involves entities that cannot be controlled by test data and predicted with sufficient precision. If these entities influence the outcome of testing, their properties must be observed and used for determining the correctness, respectively deriving expected reactions, at the time when the test models is interpreted. Typical examples are:

- Physical objects influenced by interfering, non-modeled factors or with chaotic behavior. Mocking such objects may not be sufficient for checking the behavior because the assumptions that the mock involves miss reality by far. For instance, a robot's exact movements are often hard to specify due to the impact of friction or collisions.
- Safety-critical properties whose behavior cannot be completely enforced. Mechanisms of test drivers may have some control on these properties, but the tester prefers not to risk an erroneous configuration and avoid harming the SUT or test environment.
- Technical devices with complex, state-dependent behavior. If all physical effects can be idealized, a technical device may operate conforming to an exact specification. However, if the design of this specification is complex, it may be too expensive to transform it into a test model and to execute in a simulation.
- Non-accessible properties of technical devices. Sometimes, real test data can only be gathered by connecting the SUT with productive or remote service where the tester is not in control.

In general, a system may incorporate certain details that cannot be specified within the test model and may have an impact on test outcome. As a remedy, the tester should be enabled to gather these details from additional observation from the physical world or the SASuT's interface and make testing dependent on the received information. In simulation, this information can be taken into account by accessing the respective information sources *in-the-loop* (ITL). For this purpose, the test model is executed with certain operations that query data from the interface. This data is then used for controlling the simulation.

However, for generated test-cases a more difficult hurdle constitutes. A test-case assembles a sequence of test steps, and each test step is atomic. Nothing contextualizes it except the execution order within the sequence. Stimulus models and reconfiguration automata are no longer available so that steps, where context manipulation is performed, cannot be shifted anymore. In comparison, simulation has a much larger potential to consider externally detected properties at certain points in the interpretation.

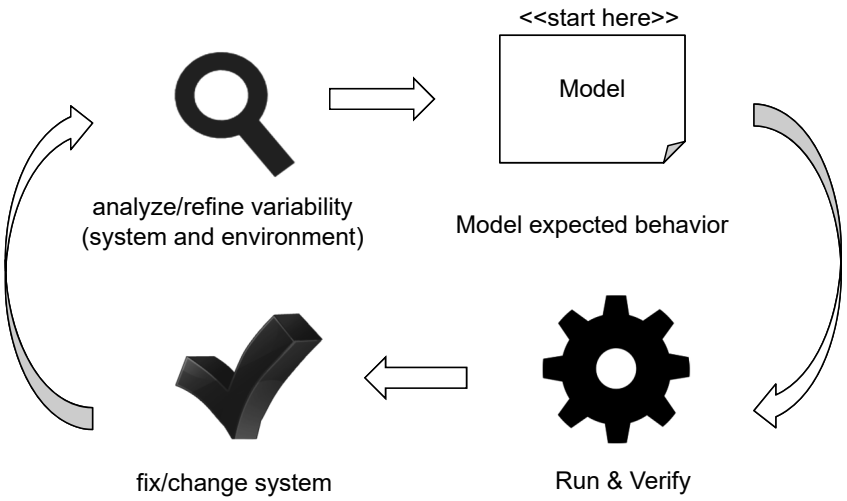


Figure 4.2.: The conceptional process of SAS testing. *The process starts by modeling the SASuT's behavior. Resulting models are run to verify the specified expectations. Failures are reported and bugs are fixed. Next, variability is analyzed or refined and the test process restarts until the quality reaches the demanded level or resources are exhausted.*

In this chapter, test-case generation and simulation in the loop shall be treated as alternatives, which are fed from the same modeling concepts but are used in different situations. If absolute reproducibility is required, generation should be chosen to create a fixed set of test-cases. Otherwise, if the SAS' correctness should be investigated in an exploratory fashion, simulation is more appropriate because it allows the tester for interacting with the SASuT and manipulating the situation at test run-time.

4.3.2. Incremental Modeling Process

From the perspective of regression testing and test-driven design [Bec03], a system version must be tested before the next one is developed. Thus, the process of designing and testing has an iterative character.

In Figure 4.2, this principle is related to the concepts, which are adopted in this thesis. We start with a standard MBT approach: a model of the expected behavior is specified, executed, and the result is reported. During the first iteration, a tester can focus on verifiable properties that are not adaption-related and remain unchanged no matter how the context appears.

During the Run&Verify phase, the model is executed, which involves test case generation or, respectively simulation, sending generated input to the SASuT, and comparing its outputs with expected ones. After fixing failures, which were found during the model execution, the test designer may distinguish coarse-grained variants of the context, adaptation modes, and respective effects on the expected behavior and outputs. With each iteration, the variants are more closely looked at and further distinguished to span a tree-like variability model. As soon as the effort for looking more deeply into variation exceeds the threshold of available computing resources or project budget, the process must be terminated.

This core process leads to step-wise refinement of the model artifacts, including its incorporated knowledge about the SAS' inherent variability. Before making use of this knowledge, in the following, the Petri net formalism is described as a base model. Later on, the Petri nets' syntax and semantics are extended to be varied by the decision points, which are derived from the found variability.

4.3.3. Basic Representation Format: Petri Nets

In software engineering, modeling concepts of different abstraction levels and degrees of formality are used. Because SAS testers face the challenge of automating test design, the executability of the underlying model types is a crucial requirement. Hence, all elements, which the model incorporates, must be completely formalized. Furthermore, because concepts of test modeling should be adopted for a broad range of implementation technologies, an adequate abstraction of these formalizations is presumed.

The majority of MDSD approaches employs MOF, UML, and OCL; the same holds for model-based test approaches. UML can be used for specifying structures of systems (component and class diagrams) and behaviors (sequence message charts, state charts, and activity diagrams). In this sense, UML is a general-purpose modeling language for object-oriented systems. But, although UML is considered a modeling standard, it is not that consistent as it would be necessary for a straight, unambiguous execution [WD11]. For instance, according to [ZH14], the exact interpretation of UML state machines is still arguable. The reason is that UML was initially designed as a more visual and conceptional language for improving the communication between engineers. To solve this inconsistency, derivatives like OCL4X [JZM07] have been proposed, which restrict UML's syntax and provide exact semantics for the remaining concepts.

Being a general-purpose language, an executable version of UML could be used for describing test models. Additional requirements as stated in this thesis—especially those, which concern verification—could only be added with UML profiles, as done in the UML testing profile (UTP, [BDG+07]). This shows that concise, comprehensible and problem-focused modeling with general-purpose languages requires additional effort. Introducing means for easing the description of self-adaption would require more extensions to UML and, thus, additionally bloat the model.

To overcome this problem, a more concise description format shall be adopted. The basis for describing the behavior that is expected from any system is a behavioral model. In the first place, the test engineer needs means for specifying how the SASuT's interface should be used for enforcing behavioral states in which certain properties are verified. As the prerequisite of this thesis does not assume white-box knowledge on the test object, the behavioral states identify certain points in the interaction protocol of the system, as seen from a functionality-exposing interface. Interactions may operate on a technical level (e.g., protocols) or on the level of dialogs controlled by a human being. To describe the outer-world interactions of SAS, this thesis employs Petri nets.

Petri nets have been first described by Carl Adam Petri in 1962 [Pet62] and are capable of modeling discrete processes in distributed systems. As syntax and semantics of Petri nets are very concise, verifications on behavioral properties like deadlock-freeness and liveness can be performed easily. Furthermore, based on this formalism, a range of extensions have been described, and each of them allows for modeling aspects of systems more efficiently but, on the other side, increase the behavioral complexity of the modeled system so that verification becomes more expensive. As Petri nets are completely formalized, their execution and, thus, test-case generation can be performed based on clear semantics. To provide modeling concepts for our specific SAS test requirements, in the following sections, extensions to Petri nets are described in detail and formalized.

These extensions will step by step be elaborated in order of the briefly announced concepts matching the SAS-specific test requirements. Each addition will follow a certain structure: (1) the introduction of a *solution together with an illustrative example*, (2) the *formal* definition of the required model elements, and—if appropriate— (3) its *usage in generation and simulation* to propose how the newly introduced capabilities are employed in the generation or simulation process.

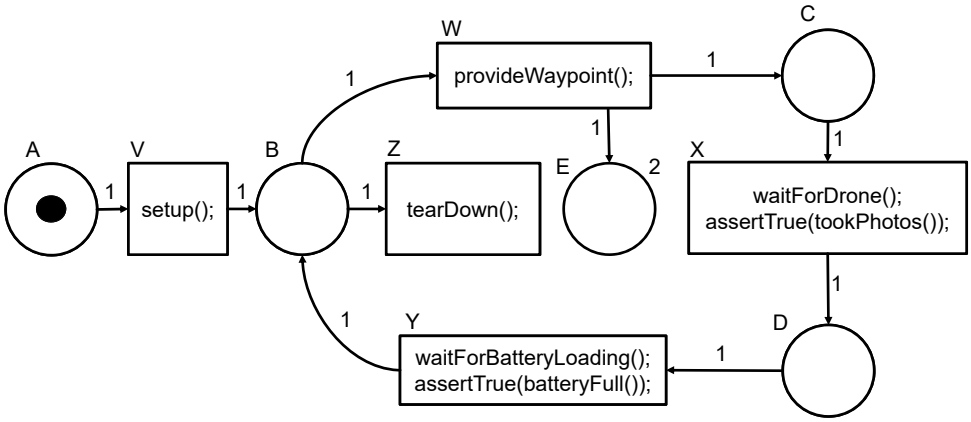


Figure 4.3.: Petri net for the drone example. *Circles are places whereas rectangular boxes are transitions. Both are connected by arcs. Dots represent tokens. Tokens are distributed in places, and each possible distribution constitutes a marking, which is a distributed state. Transitions consume tokens from incoming arcs and produce tokens along outgoing arcs. By executing a transition, the code that labels it is executed.*

Solution Concept and Example

Petri nets come with a concise graphic notation. Figure 4.3 depicts an example net, which describes the basic behavior of the introduced drone application. Informally, there are three different types of syntactic elements:

- Circles represent *places*: These elements allow for storing one or multiple *tokens* (represented by black dots). Each place is labeled with a letter $A \ldots E$ for later reference. Furthermore, a place may be labeled with a capacity that denotes the maximal number of tokens that are allowed to be stored in the place (only done for place E).

- Rectangular boxes represent *transitions*: Each transition is labeled by a letter $V \ldots Z$. Interactions are expressed by labeling the boxes with interface calls. Such operations are denoted—in this example—in a C-like fashion (i.e., operation names are followed by a number of arguments in brackets).

- Arrows represent *arcs*: Each of them either connects a place with a transition or vice versa, but never two elements of the same type. An arc can have a *weight* property; in the presented net the weight value of all arcs is 1.

Storing tokens in different places at the same time enables for modeling parallel processes with Petri nets. A state is represented by a *marking*, which relates each place with a number of tokens. More formally, a multiset can be used that is constituted by elements denoting tokens by the name of the marked place. For instance, $\{E, E, D\}$ denotes that place E contains 2 tokens and D contains 1 token while all other places are empty. The multiset can also be written in a shorter format: $\{E, E, D\} = 2E + D$. A Petri net can have an initial marking, which in this example is $\{A\} = 1A = A$.

Each transition has an *input set*, which consists of all places connected by an incoming arc to the respective transition. In the same way, the *output set* of a transition consists of all places connected by outgoing arcs. For reaching the next state, tokens from the input set are consumed as the weights of the incoming arcs prescribe, and new tokens are produced in the output set as the outgoing arcs' weights prescribe.

The Petri net's behavior is illustrated in Figure 4.4: The depicted graph represents the *reachability tree* of the given Petri net. In this tree, each node is a marking and children are reachable

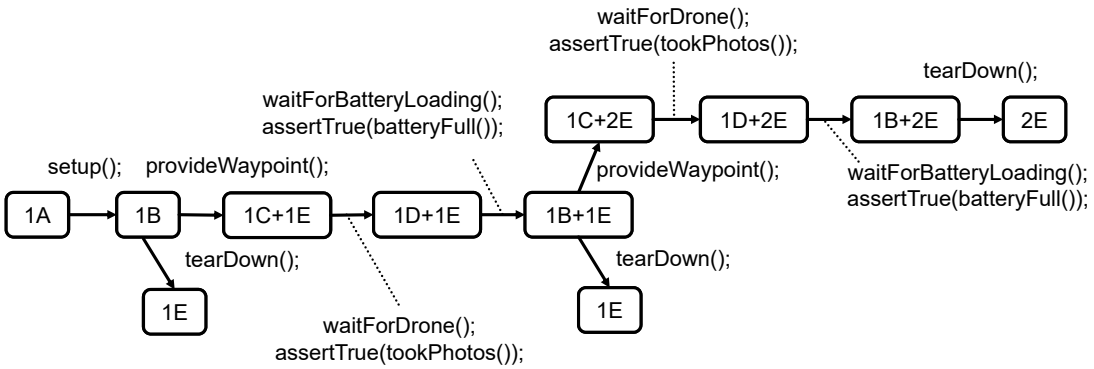


Figure 4.4.: Petri net reachability graph. *Starting from an initial marking, which is written as multi-set* $\{A\} = 1A$*, the set up transition is activated. From marking* $1B$*, two different states can be reached. Each step incorporates the execution of the respective transition's labeled code.*

markings from their respective parent. Arcs connecting parents with child nodes can be associated with transitions and, respectively, with operation labels. Due to the capacity limit of place E, the size of the reachability graph is very restricted. The tree can be constructed by searching the Petri net's state space from its initial marking, which is the root of the tree.

Execution starts with the initial marking. First, transition V is executed and the `setup()` procedure is called to initialize the drone. Second, in transition W, a waypoint is entered (`provideWaypoint()`), which has to be visited. Third, the drone starts to fly, and the model execution is paused by calling `waitForDrone()` in transition X so that the model state is synchronized with the physical SASuT. When the procedure terminates, it must be tested whether the observation data has actually been collected. For this purpose, the assertion `assertTrue(tookPhotos())` is evaluated. Fourthly, in the transition Y, the procedure `waitForBatteryLoading()` synchronizes the model execution again with a battery signal. The result of the loading process is subsequently verified by another assertion via the boolean function `batteryFull()`. Finally, a state is reached, from which the process can be restarted. Before each new assignment of a waypoint, the execution can be terminated by calling `tearDown()` from transition Z. Also, the net terminates if place `E` is marked by two tokens; this situation constitutes exactly after two iterations.

As illustrated in above example, Petri nets describe discrete behavior in a very concise manner. The exact syntax and semantics are defined in following.

Definition 4.1 (Finite Capacity Labeled Petri Net): *Formally, a Finite Capacity Labeled Petri Net is a tuple* $(P, T, F, A, L, C, W, m_0)$ *with the following components:*

- *A finite set of places* P *and a finite set of transitions* T *with* $P \cap T = \emptyset$*,*
- *a flow relation* $F \subseteq (P \times T) \cup (T \times P)$*,*
- *and finite set of action names* A*,*
- *a mapping* $L : T \to A \cup \{\epsilon\}$ *where* ϵ *is an empty action,*
- *a capacity function* $C : P \to \mathbb{N}$*,*
- *a weight relation* $W : F \to \mathbb{N}$*,*
- *and an initial marking* $m_0 : P \to \mathbb{N}_0$*.*

In general, each marking m *is a function* $P \to \mathbb{N}$ *and can be expressed as a multi-set For the purpose of describing the execution semantics, we define the input set of a transition* t *as*

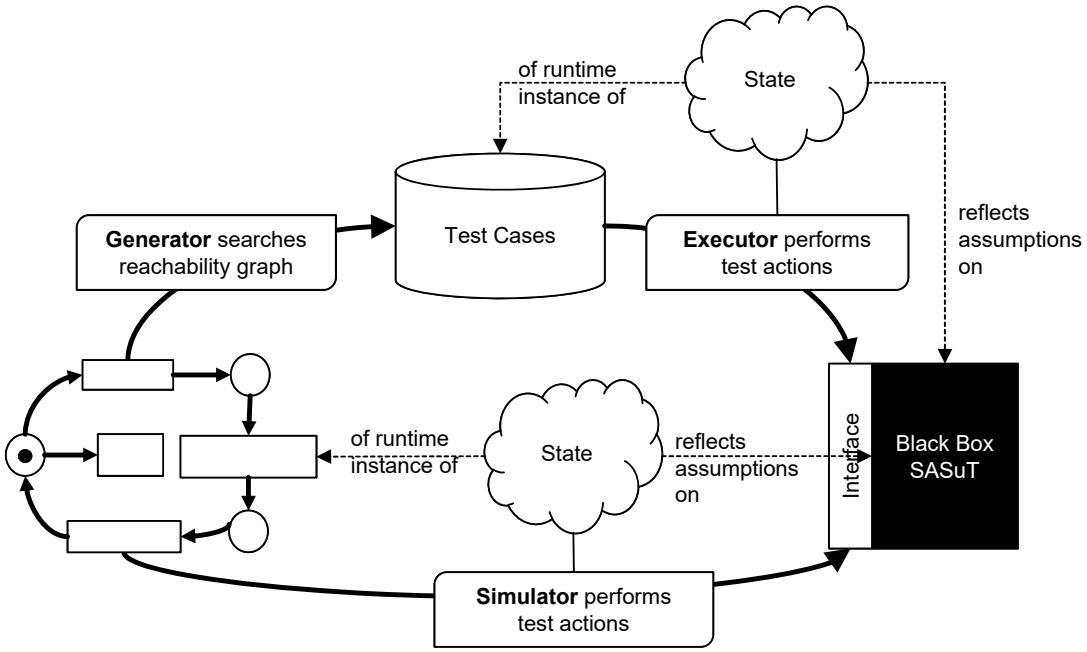


Figure 4.5.: Usage of the Petri net in simulation and test-case generation. *A generator component searches the model and produces test-cases, which can later be executed against the black-box interface. In contrary, a simulator may directly interpret the net and perform actions of a single trajectory against the interface. The test-case executor, as well as the simulator, maintain a state that reflects the assumptions on the SASuT at run-time.*

$\bullet t = \{x|(x,y) \in F\}$ *and its output set* $t\bullet = \{y|(x,y) \in F\}$. *A transition* $t \in T$ *is called* active *in a certain marking* m *if it holds the following conditions:*

$$\begin{aligned}
&\forall p_{in} \in \bullet t : 0 \leq m(p_{in}) - W(p_{in}, t) && \textit{(activation)}\\
&\forall p_{out} \in t\bullet : m'(p_{out}) = m(p_{in}) - W(p_{out}, t) + W(t, p_{out}) && \textit{(computing)}\\
&\forall p \in P : m'(p) \leq C(p) && \textit{(capacity satisfaction)}
\end{aligned}$$

Hereby, the marking m' *is one of the subsequent states of* m. *The execution of a transition* t *from one to another marking can be written as* $m \rightarrow^t m'$. *A marking, for which no transition holds all three conditions, is called terminal. In the given example (cf. Figure 4.4), the reachable, terminal markings are* $2E$ *and* $1E$. *All paths through the reachability graph from the net's initial marking* m_0 *to a terminal marking represent possible behavioral trajectories. Considering the labels of the transitions along one of these trajectories sequentially results in a test-case. Therefore, the algorithmic solution to the problem of producing test-cases from Petri nets is a depth-first search through its state space.*

□

Usage in Test Generation and Simulation

The presented Petri net model provides a completely formalized view on a black-box system's behavior. Based on its precise semantics, we can execute it automatically. As Figure 4.5 illustrates, the net can be employed for both test case generation and simulation in the loop.

In the tradition of MBT, a test modeler could automatically generate test-cases from the model. Hence, a `Generator` must be able to read the model's syntax and interpret its semantics accordingly. As a result, the reachability graph is generated, and the complete set of its paths (or, alternatively, a coverage-controlled subset) are saved as test-cases. Each test-case can be

executed by a further component, the `Executor` (i.e., a test runner). This component manages the state of the currently executed test-case and runs the test actions (i.e., the transitions' labels) against the actual system's interface. The state represents an assumption or expectation about the real system. Output values and verdicts, which occur during test execution, are stored in test reports.

Alternatively, the model may be executed directly so that the labels of its transition labels are immediately interpreted as interface operations against the black-box' interface. In this process, an interpreter plays the role of a `Simulator`. If a transition is executed, the simulator has to wait for the execution of the interpreted action. If the action succeeds (i.e., in terms of the verdict PASS), the simulation is continued; otherwise, it is canceled. Again, output values can be stored within a report. As in test-case execution, the simulation manages a state, which represents the execution of the model (i.e., its marking) and, at the same time, represents an assumption on the SUT's inner state.

Conceptually, both generation and simulation come with equal verification power in this basic setup—both approach will find the same set of failures. However, test-case generation may be better suited in regression testing while simulation can be leveraged for interaction and more exploratory testing because a tester may take control over the trajectory through the Petri net's reachability tree at run-time.

In summary, Petri nets with labels and capacity-limited places can be employed to test any black-box system with relatively low effort. However, testing an SAS only within a single environment scenario under fixed conditions is insufficient. Instead, it is more beneficial to cover several contextual configurations and let the SASuT run against them. Appropriate means for tackling this issue are discussed in the next section.

4.3.4. Context Variation

> Context variation addresses the requirements of providing sufficient expressiveness for **enforcing** and **generating** context.

Correctly functioning in different contexts is the most crucial requirement for SAS and autonomous applications. To delegate more tasks to autonomic systems, software engineers must assure the systems' quality in a large variety of situations that previously only humans could manage. Hereby, a context characterizes the situation of the system—most significantly its place, surrounding objects, and interacting persons [Dey01]. For instance, robots—including drones—and their software have to be made robust against all likely influences of their working environment. To sufficiently test an SAS, scenarios of these influences have to be varied and the system's reaction verified.

In case of the introduced application example, relevant influences are, for instance, weather conditions. From a technical perspective, such conditions are observed via sensors, which deliver streams of parameter values (e.g., temperature, humidity, brightness). The large combinatorial space of the different parameter value ranges is typically managed by employing classification and boundary analysis. Consequently, test data only covers an adequate set of combinations of representatives from each numeric range.

Besides numeric parameters, there are structural properties, such as the existence of different physical obstacles or the layout of a spatial environment, for which a set of representative scenarios have to be defined as well. Both numeric and structural properties have in common that a high number of combinations have to be examined. Due to the high number of variation points, the generative production of such combinatorial variants from models is desirable.

For representing contexts by models, a variety of approaches exist [SLP04]. These approaches can be classified as key-value models [SAW94], markup scheme models [WAP], graphical models [HIR03], object-oriented models [SBG99], logic-based models [McC93], and ontology-based

models [ÖA97]. All these representation forms are different in their level of expressiveness and, thus, the formalisms vary in the way how they support modelers to concretely analyze and design contextual concepts. In sense of SPLE (cf. Section 2.3.1), those model types belong to solution space. An additional variability model would allow for abstracting, which is always desirable in testing to classify situations to be tested by fewer representatives. Consequently, in this thesis, the modeling of context is built on concepts that stem from variability management.

In the body of knowledge of variability modeling, tree-based representations are most widely adopted. They are intuitively understood by engineers and also combine well with classification trees in testing [OMSM09]. For instance, a feature tree describes how a set of symbolic features can be combined to a valid configuration. In this way, the number of configurations can be restricted to a reasonable subset. Using certain variability mechanisms (e.g., feature-oriented programming [TKB+14]), abstract features from the problem space are mapped to the solution space so that feature selection decisions effect the configuration of the actual system. Another advantage is that trees are much easier to handle than configuration scripts. Due to these beneficial features, variability modeling for SAS testing shall rely on feature trees in this thesis.

Solution Concept and Example

Whereas features only represent binary (yes or no) decisions, contextual information may involve properties with numeric or otherwise multi-value domains. For the example application, several contextual parameters—especially weather parameters—affect the test outcome. Figure 4.6 depicts parameters, a feature tree (cf. Section 2.3.1), and classification trees for each numeric value domain. For instance, the parameter `illumnination` can be set to values of domain `ILLUM`, which permits percentual values $0 \ldots 100$. The classification tree of this domain distinguishes two equivalence classes `ILLUM_L` (low illumination) and `ILLUM_H` (high illumination). Both classes provide non-intersecting value ranges and are associated with several representatives that should be tested.

The feature tree represents configuration options of the spatial test scenario. The figure shows a 4×4 grid, where the southern lines are optional, and each line can host a predefined obstacle, which is optional as well.

Test configurations are derived by combining variants of all parametric contextual properties and variants of the given feature tree. A valid context configuration involves selected values for `illumination` (in percent, %), `wind` (in Beaufort, B), and `rainfall` (in millimeters per minute, mm/min) plus a feature configuration, which is valid against the constraints that the feature tree prescribes. Exact syntax and semantics of this contextual model are introduced in the following.

Definition 4.2 (Context Variability Model): *Feature models have been formalized in different representation formats [Bat05][CK05], each designed with appropriate properties for a certain use case. In the following, the format of Mennicke et al. is employed and, partially, simplified [MLSW14].*

As briefly introduced, variants define both properties in form of numeric parameter assignments to values from a given domain plus a feature configuration. Thus, the formal definition of the model and its configurations has to consider those two ingredients as well. We define a context variability model as

$$CVM = (F, P, V, \prec, \lhd, M, \Phi)$$

consisting of the following components:

- *A finite set of features F, properties P, value sets V,*
- *a decomposition relation $\prec \subseteq F \times F$ with $\forall (f_i, f_j), (f_k, f_l) \in \prec$: $f_j = f_l \Rightarrow f_i = f_k$ (there is only one parent per feature),*
- *a function $\lhd : P \to V$ mapping properties to value sets,*

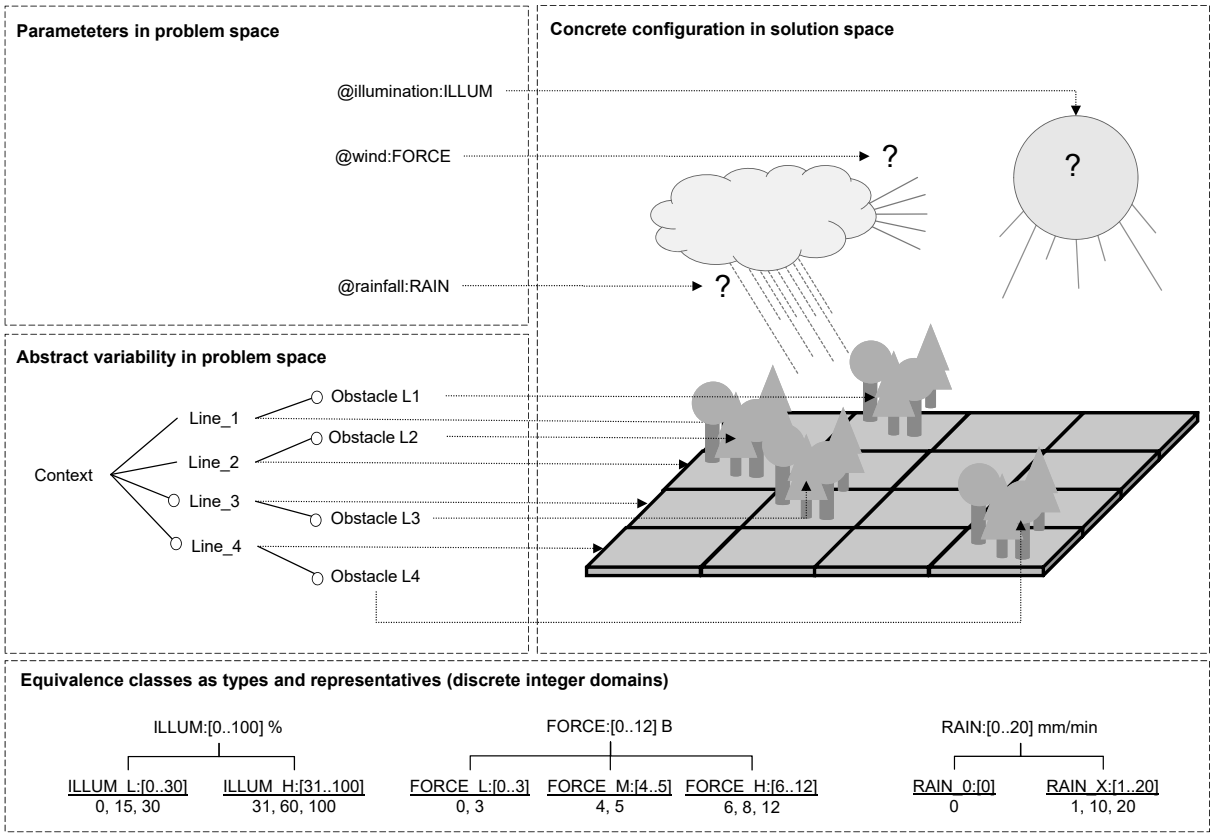


Figure 4.6.: Context variability model. *The left side depicts a set of properties for weather conditions and a feature model for terrain layout. Feature and property values can be mapped to concrete mock-up or actually arranged context conditions in the real environment illustrated on the right. The lower part shows how the property domains are further distinguished in equivalence classes with representatives to be combined in a concrete test-case.*

- *a mandatory subset of features* $M \in F$,
- *and a set of propositional formulas* Φ *over facts (atomic propositions) being features* $f \in F$ *or assignments* $(p = v)$ *with* $p \in P$ *and* $v \in \rhd(p)$.

The context variability model's semantics can be defined by mapping these components to logic propositions. For this purpose, we define the model semantically as the set of propositional formulas $\Phi_T(CVM)$ *with the same fact set as* Φ *and derived from the following rules:*

- $\forall(f_i, f_j) \in \prec: (f_j \Rightarrow f_i) \in \Phi_T(CVM)$,
- $\forall f_m \in M : \forall(f_i, f_m) \in \prec: (f_i \Rightarrow f_m) \in \Phi_T(CVM)$, *and*
- $\forall p \in P : \bigvee_{v_i \in \lhd(p)} (p = v_i) \in \Phi_T(CVM)$.

The set of valid context variants is defined as:

$$C(CVM) = C_{CVM} = \{c_i | c_i \models \Phi \cup \Phi_T(CVM)\}$$

□

The test process can make use of such a context variability model by automatically or manually deriving a valid configuration at test setup (e.g., during the execution of the `setup()` procedure). For instance, the following set represents a basic valid variant:

Example:

$$\{(illumination = 15) \wedge (wind = 3) \wedge (rainfall = 0) \\ \wedge Context \wedge Line\_1 \wedge Line\_2 \wedge ObstacleL1\}$$

The derivation process may additionally be restricted, e.g., to constrain coverage, by defining further propositional formulas. For instance, the proposition

Example:

$$(rainfall = 0) \rightarrow \neg(Line\_3 \vee Line\_4)$$

would only allow for deriving variants where spatial configurations with one of the two southern grid lines are tested with a non-zero value for the `rain` property.

Test cases, which have been produced from the previously defined Petri net, could be executed under each of the derived variability configurations. However, in this way, we only verify whether the SAS is robust enough to handle different environment situations. Besides this insight, the test engineer cannot examine whether the behavior of the system adapts as expected. Hence, he could produce different behavioral models, one for each configuration. However, such an approach is quite expensive regarding design effort. Alternatively, in the next section, an appropriate Petri net extension is proposed, which allows for modeling behavioral adaption more efficiently.

4.3.5. Modeling Adaptive Behavior

> In this section, adaptive Petri nets are introduced, which address the requirement of expressing **behavioral adaptation** in test models.

When testing SAS in a black-box setting, structural and parametric adaptation cannot always be observed. During the test process, the SUT's service interface is utilized, and the test driver depends on the resulting data flow to determine an operation's correctness. Already without contextual adaption, a system's service can be stateful—hereby the verification depends on the operations that were performed in advance. With contextual adaptation, the verification outcome additionally depends on the contextual situation. Consequently, the number of variants of context further multiplies the state space. To take this dependency into account, several extensions to the previously introduced Petri net are required.

Solution Concept and Example

To avoid risky situations, physical SAS should adapt its behavior appropriately. For instance, in case of the drone system, we expect it not to deploy or to immediately land, if potentially harmful circumstances occur, such as inappropriate weather. This expected behavioral adaption must be specified within the test model. Figure 4.7 illustrates a solution to this problem. The presented extension is based on Feature Petri Nets (FPN) by Mueschevici et al. [MCP10][MC11], who introduced *application conditions* as an annotation to transitions. Application conditions define logic expressions over features and their evaluation result affects the set of activated transitions as an addition to Petri net's operational semantics.

In the proposed model, we use a similar approach, where application conditions are logic expressions with propositions, which are defined as externalized logic functions. Thus, FPNs

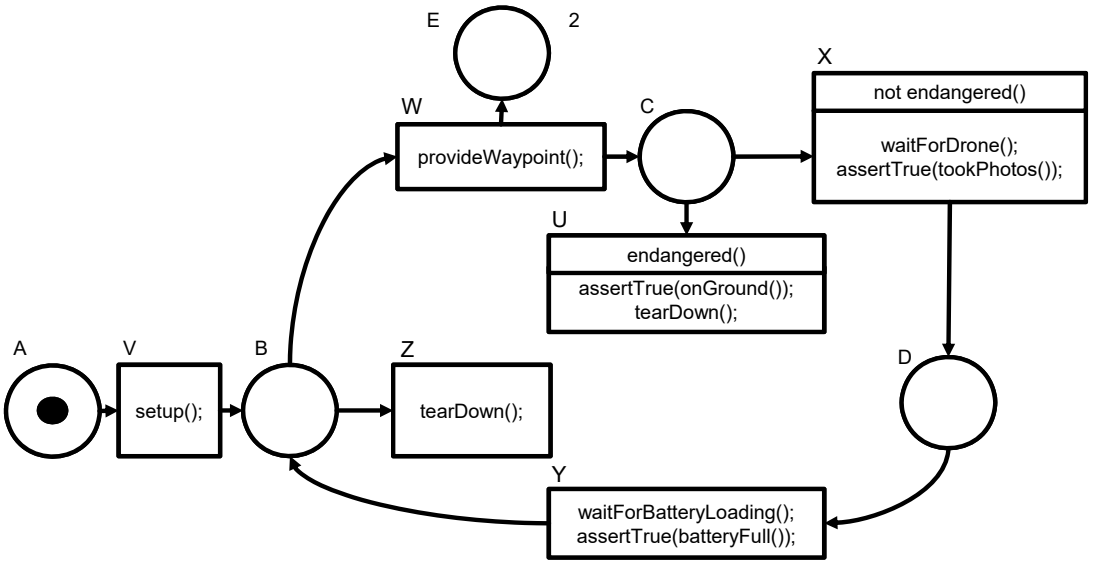


Figure 4.7.: Adaptive Petri net modeling adaptive behavior. *Additional application conditions limit the activation of transitions to situations where a certain logic function (here:* `endangered()`*) evaluates to true. These functions depend on the currently selected contextual variant. Arc weights are all set to one and, thus, omitted in the illustration.*

become more flexible because designers are enabled to externalize parts of the definition from the graphical notation. In Figure 4.7, both transitions U and X consume from place C and are annotated with an application condition. The expression restricts the activation of a transition to states (i.e, markings) where the logic function `endangered()` evaluates to `true`, respectively to `false`. By defining this function as a proposition over contextual features, the control flow can be steered depending on the selected context variant. To perform such a selection and configure the environment, we assume that the `setup()` procedure in transition `V` selects an initial variant.

During Petri net interpretation, the selected variant is taken as fact set for the evaluation of application conditions. For instance, the logic function could be defined as follows:

Example:

$$endangered() \mapsto (wind = w), (rainfall = r) \in C(CVM) \\ \wedge\, w \notin FORCE\_L \vee r \in RAIN\_X$$

In other words, `endangered()` evaluates to true if there are assignments to the properties `rainfall` and `rain` with values not in equivalence class `Force_L`, respectively values in class `RAIN_X`. In this way, logic functions on the current context variant can be specified and behavior can be defined based on these functions. Depending on the test method, the evaluation of application conditions may happen synchronous to the execution of the SASuT (i.e., during simulation) or at a completely different time (i.e., at generation time of test-cases).

In the given model, the adaption is *static* because it is performed during `setup()` before the actual operation starts. The procedure expects one single valid environment situation to be selected from the modeled variant space. An adaptation that is performed by the SASuT cannot be directly investigated from outside the black-box. The test system observes its behavioral effects and verifies them by inspecting outputs of the service interface.

For FPNs, two alternative semantic formalizations are provided by Muschevici et al. [MCP10] One is based on projecting the net on transitions that are still applicable after configuring a

certain variant. This solution can only be used if the configuration is not considered to be changed during the net's run-time. The second formalization is based on operational semantics, which lets the syntactic structure of the net untouched but defines the execution of a transition dependent on the evaluation of its application condition. Thus, also a dynamically altered configuration state can be considered. As reconfiguration establishes the actual power of self-adaptivity, the latter semantics variant is the only reasonable alternative for test-modeling SAS. Based on the operational semantics, in the following, a formalization is given, which extends FPN by additional functions over context variants. In this thesis, the resulting model shall be named *adaptive Petri net*.

Definition 4.3 (Adaptive Petri Net): *As prerequisite, we define the following components:*

- *A context variability model* $CVM = (F, P, V, \prec, \lhd, M, \Phi)$,
- *its variant space* $C_{CVM} = \{c_i | c_i \models \Phi \cup \Phi_T(CVM)\}$,
- *the set of all possible logic functions over variants* $\Lambda = C_{CVM} \rightarrow \{\top, \bot\}$,
- *a grammar for application conditions*

$$g ::= f \mid \lambda \mid h \mid g \texttt{ and } g \mid g \texttt{ or } g \mid \texttt{not } g$$

$$h ::= a \texttt{ == } v_a \mid a \texttt{ > } v_a \mid a \texttt{ < } v_a$$

  where $f \in F$, $\lambda \in \Lambda$, $a \in A$, $v_a \in \lhd(a)$, *and* G *is the set of all possible application conditions,*

- *the Finite Capacity Labeled Petri Net* $PN = (P, T, F, A, L, C, W, m_0)$,
- *and, finally, a relation between transitions and application conditions* $\Sigma \subseteq T \times G$.

The resulting adaptive Petri Net is a tuple $VPN = (PN, CVM, \Sigma)$. *As for finite capacity labeled Petri nets, adaptive Petri nets also require a definition of semantics. Here again, we need a prerequisite—the definition of the satisfaction of application conditions. We define that a variant* $c_i \in C_{CVM}$ *satisfies an application condition* g *in the following cases* ($c_i \models g$):

$c_i \models f$	*iff*	$f \in c_i$
$c_i \models \lambda$	*iff*	$\lambda(c_i) = \top$
$c_i \models g_1$ `and` g_2	*iff*	$c_i \models g_1 and c_i \models g_2$
$c_i \models g_1$ `or` g_2	*iff*	$c_i \models g_1 or c_i \models g_2$
$c_i \models$ `not` g	*iff*	$c_i \not\models g$
$c_i \models a$ `==` v	*iff*	$a = v \in c_i$
$c_i \models a$ `>` v	*iff*	$\exists v = b \in c_i : b < a$
$c_i \models a$ `<` v	*iff*	$\exists v = b \in c_i : b > a$

Furthermore, to complete the definition of a adaptive Petri net's semantics, we add the following condition besides the already stated ones (activation, computing, capacity satisfaction; cf., Definition 4.1):

$$\Sigma(t) \models c_i \quad \textit{(application condition satisfaction)}$$

□

Based on the proposed model, testers can model an SASuT's expected behavior depending on a contextual variant, which is established before testing. To leverage these capabilities to a point, where dynamic adaptation is available to test modeling, the next step is to define permitted context changes to be tested.

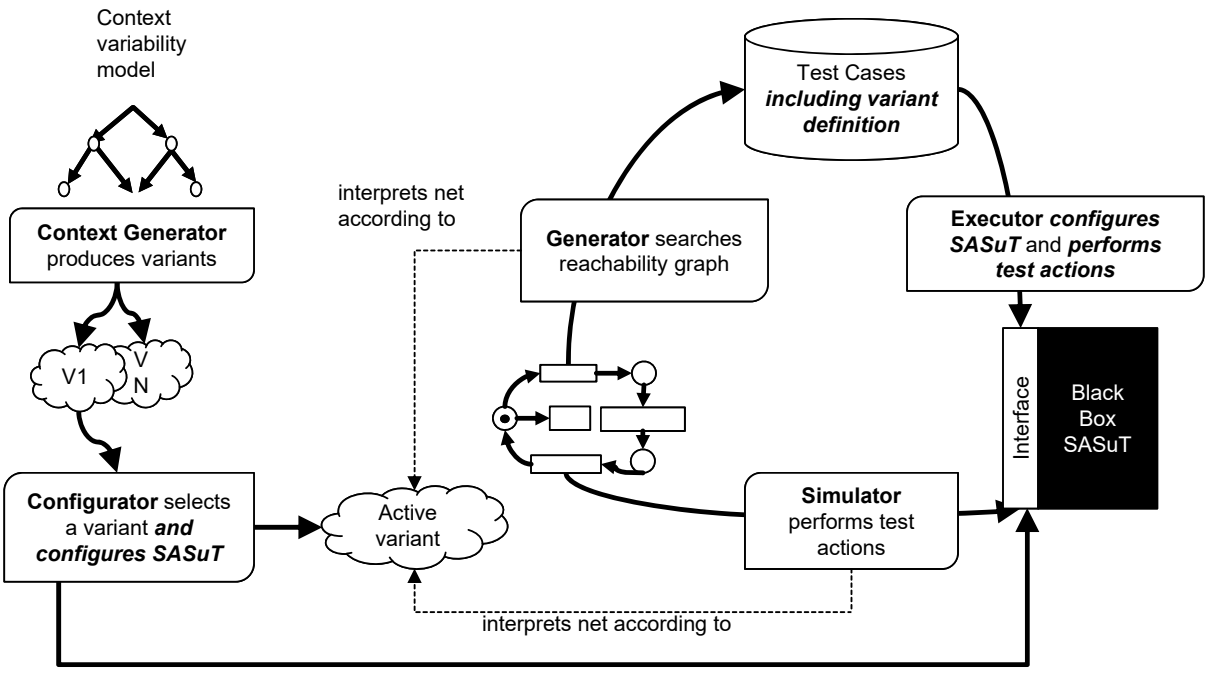


Figure 4.8.: Usage of the variability model in simulation and test-case generation. *From the variability model, several context variants are derived automatically or manually. At the beginning, the configurator selects one of these variants. Subsequently, the adaptive Petri net is executed according to the propositions of this active configuration.*

Usage in Test Generation and Simulation

The integration of the context variability model and the adaptive Petri net into the proposed test infrastructure is depicted in Figure 4.8. An extra `Context Generator` produces variants from the given context variability model. Alternatively, a test engineer may manually pre-define some variants, which can be validated against the variability model. In contrast, the automated derivation requires the mapping of the variability model to propositional logic statements or constraints, which are input to a satisfiability (SAT) or constraint solver. Each found solution is a variant `V1` ... `VN`, which consists of features and value assignments.

Another component is the `Configurator`, which selects one of the produced variants so that the `Generator`, or, respectively, the `Simulator` can interpret the adaptive Petri net according to this configuration. This process can be performed multiple times to generate or simulate new scenarios. For each variant, a new simulation may be initialized, or a new test-suite with different context-dependent test-cases is generated.

However, it is not sufficient to just set the context state for model execution. The selected variant must also be configured in the physical test bed. Either respective mock-ups must be employed to virtually simulate another situation, or the environment is manually or, if possible, automatically adapted for the selected variant. In such a case, an interpretation of the variant (i.e., a mapping to solution space) must be performed by using test-automation. For our example, we assume that a technical component exists, which can translate the variant propositions to respective set-up operations on the test bed.

The point in time, when the context variant is configured, differs in test-case generation and simulation. Whereas in the latter case the selected variant can directly be deployed—hence, we use the `setup()` procedure—in generated test-cases it has to be performed just before executing a test-suite. In consequence, the variant that was input to the generator has to be persisted with the associated test-cases. Before starting test-case execution, the `Executor` deploys this formally associated variant. For both simulation and generation, the test report should be contextualized

with the active variant as well; otherwise, important information for the later evaluation is lost.

In summary, variability management for a tested context space can be integrated with the behavioral definition by extending the Petri net's formalism. But still, only those SAS can be exhaustively tested, which solely adapt at deployment-time. To overcome this hurdle, the following sections present concepts for controlled manipulation and verification of adaptation at run-time.

4.3.6. Dynamic Context Change

> Modeling dynamic context change addresses the requirement of expressing the **stateful change of context**.

SAS outperform traditional systems because they continue working properly even when environment conditions fundamentally change. Of course, this claim has to be verified by testing the SAS under forcefully changed contexts. In the sense of the concepts provided in the previous chapters, this can be reached by altering the variant derived from the context variability model. The basic principle is to derive a set of variants and change between them during the Petri net's execution time.

This task requires for additional expressiveness within the employed context model. The tester must be equipped with concepts to define when and how a context variant is switched to another. Different scenarios are possible how to specify this dynamic change: lists of variants, reconfiguration rules, or even full-fledged operational logic. Depending on the concrete appearance of the tested system, one solution might be more efficient than another. Consequently, the following proposal shall not introduce a concrete model but concepts and an interface, which concrete models of contextual change can adhere to.

Solution Concept and Example

The decision, which variant follows another in testing, could be random, heuristic, or statistic as done in DiVA [MBS10]. However, whereas such discriminators are indeed capable of creating a reasonable coverage on traditional systems' test models, they are not sufficient for exhaustive testing of SAS because they miss some adaptation-specific requirements, which were discussed briefly. Firstly, the adaptations of the system can be stateful, i.e., one adaptation mode depends on the previous ones. For instance, the drone system may observe weather values over a longer timespan and may be designed to adapt when a value's (e.g., an increasing rainfall) tendency aims at a critical value. To test this scenario, the selected context variants should be enforced in the correct order. Another example is a hysteresis, such as a built-in control loop of electrical systems. Furthermore, there may also exist software or other technical systems that maintain an inner state, which cannot be reset each time of reconfiguration.

Secondly, testers may want to configure more representative than untypical scenarios in testing, as the likelihood of being confronted with latter ones is lower and, thus, not that significant for the user' acceptance. For instance, the drone system is exhaustively tested at daylight, but only a minor set of test-cases will be produced for night conditions to verify that the drone denies flying without sufficient light conditions.

Especially for contexts with a large state space, a more effective mechanism for controlling context change is needed. To create meaningful change scenarios through the variety of context states, in this thesis, the usage of different domain-specific models is proposed. Figure 4.9 illustrates this concept. Based on the context variability model, the considered configuration space is defined (upper part). Sub-trees of the feature model in combination with representatives of property's value classes can specify variability in certain distinguishable domains. For the drone scenario, these may be landscape properties, weather conditions, and any urgency level.

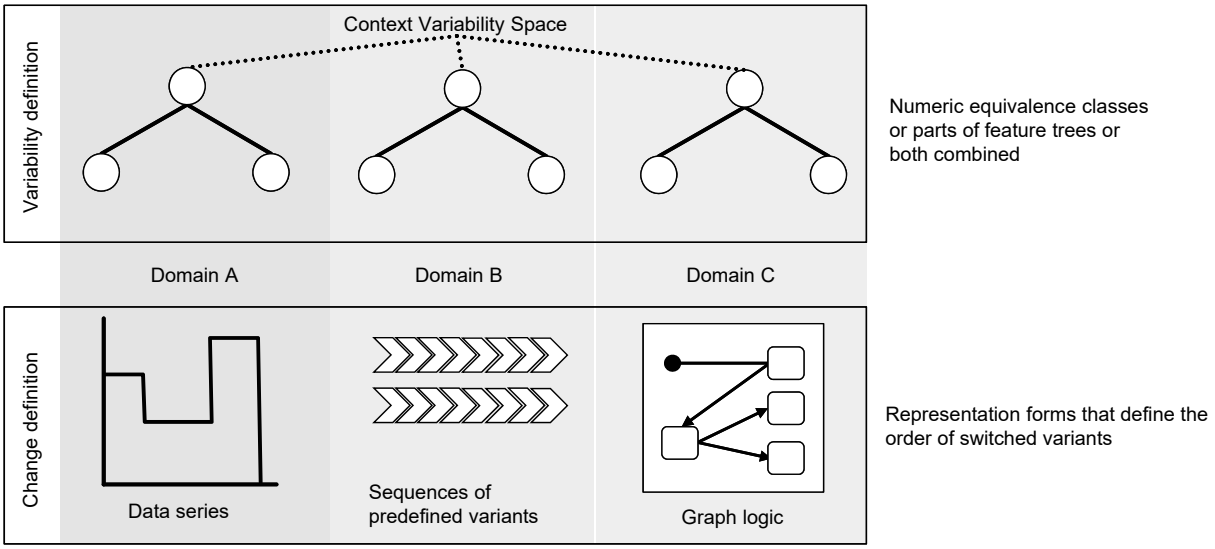


Figure 4.9.: Different types of change models.. *Dynamic contexts can be represented in different forms. Parts of the variability space may be covered by data series, others the predefined sequences, or graph logic.*

Each of these domain-specific configuration properties changes differently. So, for instance, weather values change continuously, whereas the change in technical systems is triggered by events.

A variety of representation forms for such change definitions exist. For interactive systems, which the SAS communicates with, and which contribute to its context, a finite state machine may be the model of choice. For physical and numeric properties, data graphs that represent data series or a numeric function are useful. Others could rely on a fixed set of predefined scenarios. For spatial dynamic variability, a movement profile could be provided.

In consequence of this heterogeneity, it is difficult to provide test modelers a fix and generic toolset for defining representations for their specific domain, which work with our previously provided concepts. To overcome this problem, in the following, only prescriptions on the interface of such models are given. This interface shall allow for specifying which inputs and outputs a model must be able to consume or produce without limiting its exact appearance. The unified usage of such models requires the following premises concerning their interface.

Context Variant Manipulation Actions The adaptive context model has to be capable of changing the active context variant to another. Hence, its semantics must be formalized as *manipulations on the context variant* within the given variant space. There are several options how to achieve that. Firstly, the model could switch between a pre-computed set of variants. Because the set only contains validated variants, this approach is safe because no invalid variants can be reached. Secondly, special actions could be employed, which change the active variant to reach another. Here, the problem constitutes that modelers are enabled to manipulate a variant and reach an invalid one. This inconsistency can be handled by either checking the test model for all reachable variants and notify the engineer if he made such a mistake, or, alternatively, the interpreter semantics can be defined in a way so that the execution is canceled as soon an invalid variant is reached. The latter method is questionable as it requires either the complete pre-computation of all valid variants (potentially extremely inefficient) or checking the validity of each reached variant at execution time (which potentially lowers the simulation performance).

However, in this thesis, we make use of the concept of change actions because they provide maximal flexibility for describing reconfiguration. As the context variability model provides only two fundamental concepts—features and property value assignments—only change actions for

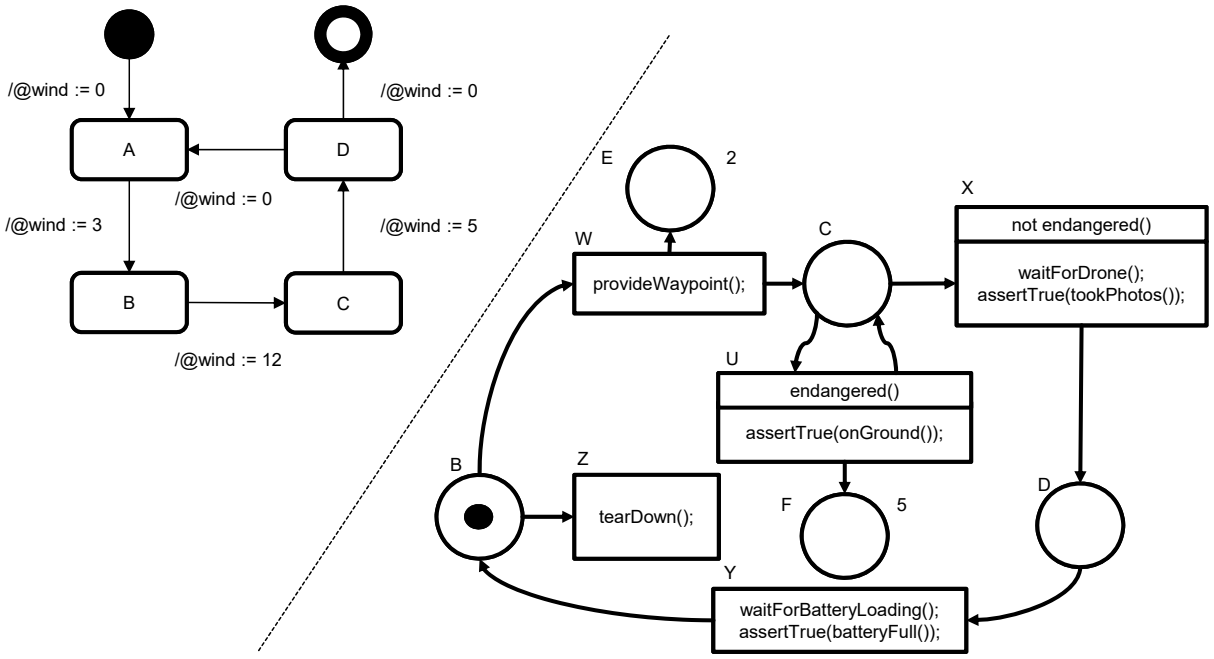


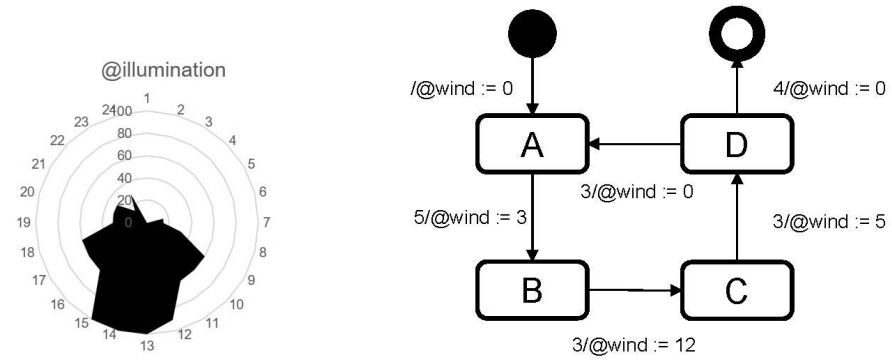
Figure 4.10.: Variant manipulation and its effect on the definition of expected behavior. *The model on the upper left manipulates the wind property value which influences the evaluation result of* `endangered()`. *In this way, the adaptive Petri net's behavior is affected. A produced test-case is a sequence of different interleavings of context manipulations and Petri net traces.*

those two have to be defined. In case of features, actions must be provided, which `add` or `remove` them from a variant; for properties, a `reassignment` action is required. For instance, in the drone case, obstacles may be added or removed during test execution or the brightness value may be reassigned. Based on these three actions types, the change of the active variant and the expectations to the SASuT can be specified completely.

Figure 4.10 shows an example. The Petri net has slightly changed so that the transition for `U` is connected to place `C` in both directions. Because there is a new place `F` with a maximal capacity of 5, no more than five such checks will be performed during the complete execution. To change the outcome of the functions' evaluation, a finite state machine is introduced, which performs reassignments when its transitions are executed. The initial variant has a wind property value of 0 and is eventually set to 3 at a later point in time. In the latter situation, `endangered()` will evaluate to false, which changes the execution of the adaptive Petri net.

The approach allows for changing the variant in parallel (i.e., interleaved) to the actual behavioral test model and influences the latter one's execution. In this way, the tester can model dynamic contextual change and its expected effects on an SASuT's behavior. Due to the interleavings, the state space grows combinatorially if no further restrictions are applied. Despite this growth, the model limits the variant space, because not every representative of the predefined equivalence classes may be reached. For instance, in the given wind-related state machine, the value 4 (of equivalence class $FORCE\_M$, cf. Figure 4.6) is never assigned.

For the representation of the dynamic change of wind measures, a state machine is used in the given example. Potentially it could switch and produce test data infinitely many times. Another important characteristic is that it must be initialized by setting the modeled initial state A. In this way, also all other dynamic context models could contribute to an initial context variant. Potentially, their order of activation could play a role if we allow for manipulating them intersecting portions of the variability space. Another option would be to define a static, initial variant, which is considered before any activation of any dynamic context variant.



(a) A circular mapping of virtual time to values of the illumination value space.

(b) This model represents how to set the wind value based on virtual time control within a state machine.

Figure 4.11.: Virtually timed change models.

Budgets of Virtual Time as Control Mechanism The introduced modeling concept for defining context dynamics are open to be used with different representation forms, such as data series, state machines, and movement profiles. The heterogeneity resulting from this flexibility raises the question of how a test modeler should control the interleaving, order, and timing of context changes with each other and with steps stemming from a contextualized adaptive Petri net. For this purpose, we introduce a uniform model control by *discrete budgets of virtual time*. The source of control shall be a clock for now, which is external from the actual stimulus model.

For instance, two possible virtually-timed models are depicted in Figure 4.11. In Figure 4.11a, a model for the `illumination` property is shown. It is a circular graph, which can produce potentially infinite-long scenarios. In consequence, after 24 time steps, a new day begins and each day gives the same values. Alternatively, there could be a mechanism in the model's semantics, which resets the time after each day or the property value remains complete untouched (i.e., constant) after one single day.

In Figure 4.11b, the previously introduced finite state machine for wind is extended with triggers of time budgets, whereas the initial setup is executed immediately. Thus, the initial assignment is activated without waiting while all other transitions accept a specific amount of virtual time. When the terminal state is reached, the property value of `wind` is not altered anymore.

Whereas the circular model is completely deterministic, the given state machine is not. Hereby, the non-determinism resulting from the usage of virtual timing exceeds those resulting from transitions. Assuming state `D` is reached and an additional time budget of 4 is added, both transitions from `D` become executable. Whereas the transition to the final state would completely consume the provided budget, the transition the `A` would not. The latter behavior depends on the precise semantics of the stimulus model, what is done with the remaining budget—it could either be neglected or accumulate and added to the next incoming budget. Viewing stimulus models as real-world representations of processes in the environment, the latter solution promises more consistency. Thus, budgets are accumulated over time and each transition with a lower or equal accepted time value than this accumulation can be activated. For models that consume all budgets immediately (as the illumination model does), a *reservoir* of accumulated time has to be maintained.

Furthermore, it may also be the case, that a time budget is produced, which exceeds multiple executable transitions' budgets so that one of them has to be chosen. Even after the execution of a transition, there may be enough remaining accumulated budget to execute a next one. For instance, if we give 16 portions while being in state `A`, a whole round back to `A` is performed and still 1 portion of time is left. Another effect is that due to the optional execution of a transition

after its time budget is available, it also may never be executed. As this behavior produces a vast expansion of the whole model's state space, the effect should be avoided by defining semantics in which always the maximal budget-consuming transition is activated as soon as possible, which we assume hereby.

Another considerable point is that time budgets of different change models must be normed against each other. If it can be assumed that the model for `illumination` takes hours as input, this does not have to be valid for the `wind` model.

Based on virtual time, all domain-specific models can be controlled in a universal manner. The definition of how virtual time progresses can be steered from a predefined clock, which is started with the simulation process.

Reflecting Changes of the Test Model by Respective Actions on Test Bed Synchronously to variant reconfiguration, the model has to define test actions that *update the SASuT's actual context* in parallel to the model state to enforce the respective situation. As the active variant reflects this context state, both must be changed in parallel. There are several options how to do that. For instance, in dynamic delta modeling [Hel12b], each manipulation of a system (i.e., a delta) is related to respective features. In this approach, the dynamic reconfiguration is defined by an automaton where states are variants and transitions carry guards (application conditions) about features, as well as deltas on the real product. Hence, it is necessary to explicitly define variants and, thus, to pre-compute these. To avoid the effort of this computation, the proposed model uses a certain set of interface actions that manipulate the context via a test driver.

Models complying to all of these three presumptions shall be called *stimulus models* because they not only manipulate the variant considered by the model interpreter but also stimulate the actual test environment to keep it synchronized with the interpreter's assumptions.

Definition 4.4 (Stimulus Model Interface): *In conclusion, there is a set of alternatives how contextual change can be described. The requirements for these alternative models as defined above can be formalized as a common interface. Hence, we define a set of possible change operations on the variant model. With* $CVM = (F, A, V, \prec, \lhd, M, \Phi)$ *being a feature model, the language* Σ_o *consists of the following literals:*

- *Activation operations* $+f$ *with* $f \in F$,
- *deactivation operations* $-f$ *with* $f \in F$, *and*
- *assignment operations* $a_i := v$ *with* $a_i \in A$ *and* $v \in v(a_i)$.

Each context stimulus model over the variability model CVM has to define a relation of the following type:

$$\lambda^{CVM} \subseteq \mathbb{V} \times \mathbb{N} \times \mathbb{N} \times \Sigma_o^* = \{(V_s, t_c, t_a, w_o) | V_s \in C_{CVM} \wedge t_c, t_a \in \mathbb{N} \wedge w_o \in \Sigma_o^*\}$$

The tuple component V_s *is an initial variant,* t_c *is a budget to be consumed,* t_a *the accumulated budget after the operation and* w_o *a word of change operations.*

The introduced relation maps to actions instead of to resulting variants so that concurring, interleaving changes from multiple stimulus models are avoided. The semantics of a change operation is defined as follows:

$$\begin{array}{lll} v_i \xrightarrow{+f} v_j & \textit{iff} & v_j = v_i \cup \{f\} \wedge v_i, v_j \in C_{CVM} \\ v_i \xrightarrow{-f} v_j & \textit{iff} & v_j = v_i \backslash \{f\} \wedge v_i, v_j \in C_{CVM} \\ v_i \xrightarrow{a:=val_j} v_j & \textit{iff} & v_j = \{a = val_j\} \cup v_i \backslash \{a = val_i\} \wedge v_i, v_j \in C_{CVM} \end{array}$$

□

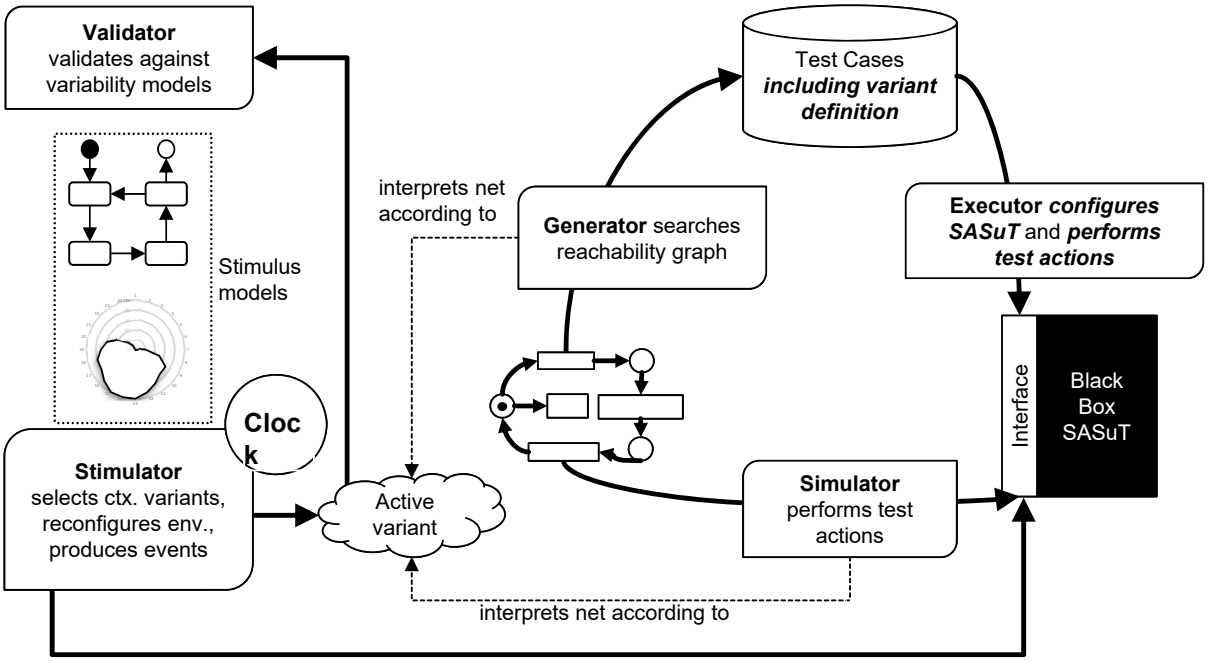


Figure 4.12.: Integration of stimulus models in simulation and generation. *Instead of just performing a configuration at a single point in time, the stimulator interprets the given stimulus models, controlled by an external clock, and, thus, changes the active variant. In this process, also operations at the test bed's interface are performed to manipulate the actual test environment.*

Usage in Test Generation and Simulation

Stimulus models can be integrated into the simulation, respectively generation, infrastructure by leveraging the configuration component to a run-time `Stimulator` as presented in Figure 4.12. The stimulator manages and controls the active variant based on the operations that the stimulus models define. To trigger such a contextual reconfiguration, an external clock has to employed for producing virtual time, which is consumed by the stimulus models. As each triggered reconfiguration potentially maps to a manipulation action on the context management interface, these actions have to be propagated at run-time as well. When using reconfiguration actions to reach a new variant, we replace the context generator component by the `Validator`, which checks the reached variant at run-time. If the validation fails, the models are incorrect, and the relevant path will be discarded in generation or, respectively, the simulation stops.

4.3.7. Interfacing Context from Behavioral Representation

> Interfacing and synchronizing stimulus models and adaptive Petri nets tackles the requirement of expressing **stateful change of context** with a special focus on **state space limitation**.

In its current form, the generation and simulation infrastructure maintains two partially independent control flows. Firstly, the adaptive Petri net is searched for paths, which depends on the selected context variant. Secondly, context change works on its own, only driven by an external clock. This setup results in an infinite number of possible interleavings between context changes and Petri-net-controlled, behavioral steps because each behavioral state would potentially be tested in any reachable context variant.

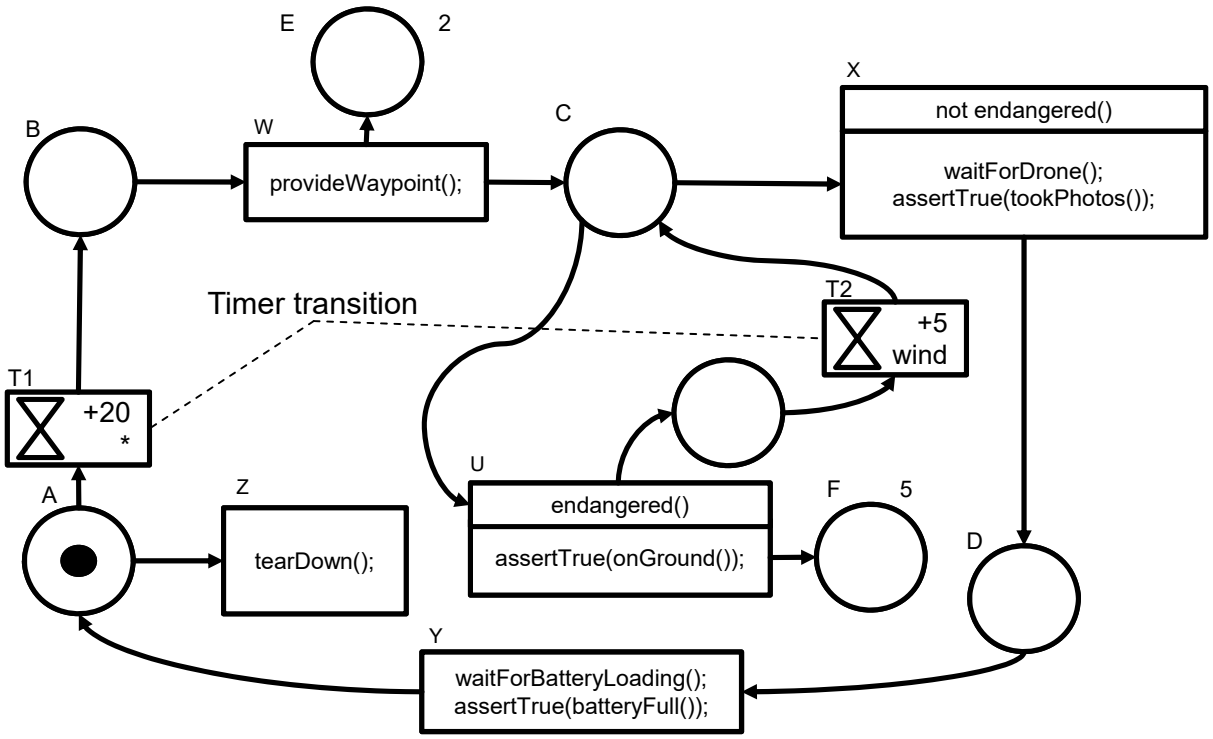


Figure 4.13.: Adaptive Petri net with timer transitions. *Each timer transition comes with a definition of the time budget to be produced plus a list of stimulus models (or a wildcard targeting all models) to be triggered.*

In contrast, testing aims at reducing state space and only verify correctness in risky situations because in most systems no complete coverage is possible. Hence, in this section, new elements are introduced, which enable the test modeler to backward-synchronize adaptive behavior with context change. Such synchronization operates against the actual direction of the control loop, outgoing from an adaptive Petri net as the behavioral description to context dynamics. The core benefit of this approach is that it allows the test designer for defining in which behavioral states new contexts should be tested. Thus, the number of possible interleavings can be drastically reduced to a manageable amount.

Solution Concept and Example

For backward-synchronizing adaptive Petri nets with stimulus models, we construct a new type of transition— a so-called *timer transitions*—, which can deal as a model-inherent clock without actually changing the adaptive Petri net's operational semantics. To take control outgoing from the Petri net, those timer transitions produce amounts of virtual time in the moment of their execution. The production has to be *atomic* so that no ambiguous interference with the execution of another transition constitutes.

The principle of how timer transitions behave is illustrated in Figure 4.13. Between place A and B, the new timer transition (denoted by an hourglass) $T1$ adds 20 units of time to be consumed by stimulus models. Besides the numeric definition of time budgets, each transition of this type points to one or multiple stimulus models, which are fed from this point of operation. Whereas transition $T1$, which is marked with a wildcard ($*$), triggers all defined stimulus models, transition $T2$ only triggers the stimulus model for the wind variable. If multiple stimulus models should be triggered, we assume that they change the contextual state in order of their definition, which avoids non-determinism. In an adaptive Petri net that incorporates timer transitions, context change actions and context reconfigurations are aligned with the Petri net's control flow.

Definition 4.5 (Timer Transition): *Additional to Definition 4.3 of adaptive Petri nets, we define timer transitions syntactically by altering the the label mapping L to*

$$L_{Time} : T \mapsto A \cup \{\epsilon\} \cup (\mathbb{N} \times SM^*)$$

where $SM = \{\lambda_i^{CVM} | i \in N\}$ is the set of known stimulus models (represented as their transition relations under the feature model CVM) and SM^* words (i.e., lists) of stimulus models, which determines the order of their execution. Besides actions and empty labels, now pairs of time budgets and subsets of known stimulus models are permitted. Only, transitions that are labeled with such a pair are classified as timer transitions. Consequently, a finite capacity labeled Petri net with timer transitions has the format

$$PN_{Time} = (P, T, F, A, L_{Time}, C, W, m0)$$

and, respectively, an adaptive Petri with timer transitions has the format

$$VPN_{Time} = (PN_{Time}, CVM, \Sigma)$$

.

When a timer transition is activated, the stimulus models are operated in the order of their definition and according to the semantics of definition 4.4. Whereas labels that only contain operations on the test system are uncritical for the semantics of the Petri net, a timer transition alters the contextual state and, thus, influences the further execution.

□

Usage in Test Generation and Simulation

In the last version of test infrastructure, an external clock was necessary to operate stimulus in parallel to adaptive Petri nets. This clock is now replaced by the mechanism that timer transitions provide. Figure 4.14 depicts the changed control flow. During the execution of the behavioral model by the generator or simulation engine, time budgets are delegated to the context simulator, which then alters the context respective to the relevant stimulus models. After a reconfiguration has been performed, the next execution step of the execution of the adaptive Petri net potentially operates in a new context so that a causal loop is established. Within this loop, the Petri net's representation is the central point of control and its included timer transitions align its progress with the contextual simulation.

With the given infrastructure, concepts, and models up to this section, all points in the declared SAS state space can be tested. However, still, there is no explicit means for adaptation modes. Instead, as Figure 4.14 illustrates, an adaptation mode is an implicit state during interpretation, whereas the relation between context and adaptive behavior is direct. To gain a better separation between the state of context and the state of adaptation, in the next section, an extra variant space is introduced.

4.3.8. Adaptation Mode Variation

> The in the following elaborated approach to explicitly vary adaptation modes addresses the requirement of **parametric adaptation**.

Making not only the context variant but also the adaptation mode explicit creates additional reuse. In the previous chapters, we assumed that only the context's state is modeled explicitly within stimulus models, but not which system variant that is selected during the resulting reconfiguration. In stimulus models, actions are used to enforce that a certain state is reached and to verify that the SASuT behaves correctly. However, it is not completely appropriate that

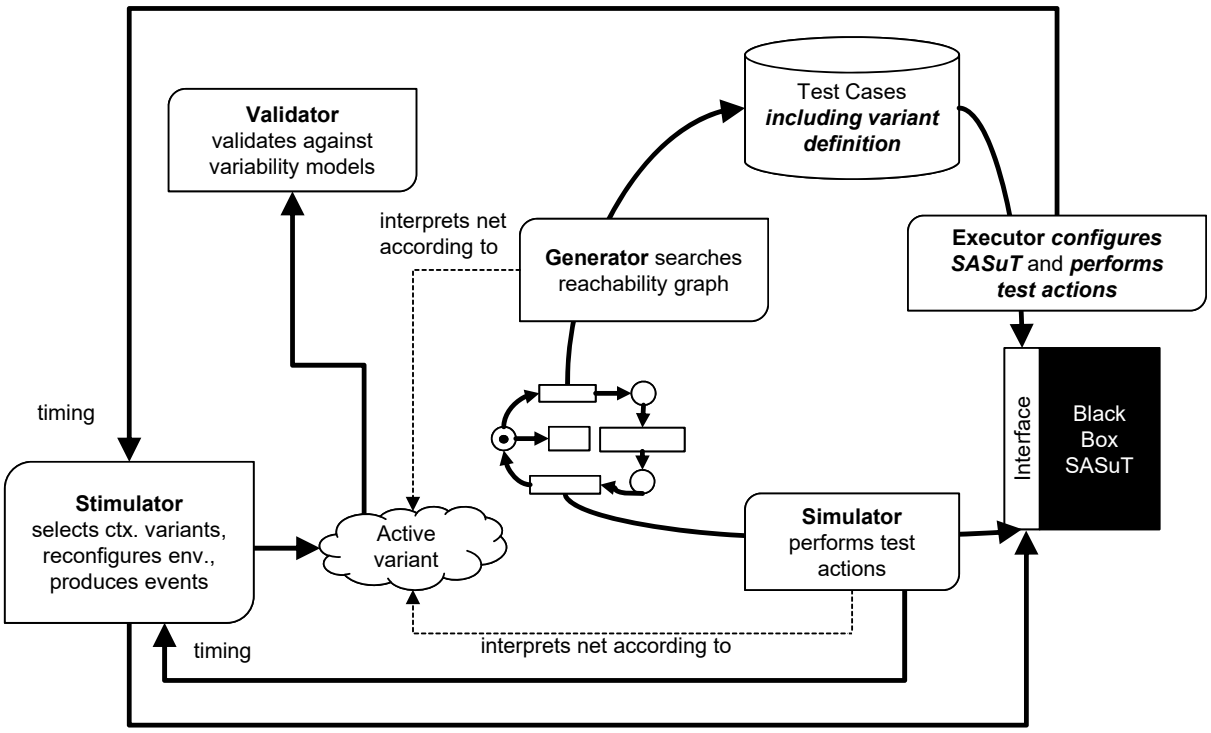


Figure 4.14.: Generation and simulation infrastructure with timer-transition-based synchronization. *Instead of using an external clock, the time budgets are now generated from the adaptive Petri net models. Thus, the contextual change can be controlled from the central Petri net specification.*

system behavior is defined directly dependent on context situations. With a closer look at the concept of SAS adaptation loops, the causal chain between context analysis and adaption can be separated in different model artifacts. In this manner, we model context and its effects on the SAS's adaptation mode, on which the tested behavior depends. Thus, a separate consideration of context and the resulting SAS adaptation mode is beneficial to decouple both better and to define their causal connection explicitly.

Basically, both context variant and adaptation mode reflect variability in two different dimensions of the SASuT. In SPL design, it has been discovered that it can be beneficial to model certain dimensions of variability in isolation. In [LK10], Lee and Kang distinguish three such dimensions: (1) variability of usage context (UC) defining contextual settings in which a product is deployed or used, (2) variability of quality attributes (QA), which a product must satisfy, and (3) variability among product features (PF) describing a set of capabilities that can be provided by the product. These dimensions are independent but can be mapped to each other for all three combinations (UC-QA, UC-PF, QA-PF). An UC-QA mapping relates feature sets of the UC domain to the QA domain. Subsequently, weighed mappings (from ++ to −−) between QA features and product features can be established depending on how much effect a quality attribute has on the adequacy of a product feature. Direct mappings form UC to PF state which product features are required or excluded in relation to a UC feature.

A more comprehensive model is proposed by Rosenmüller et al. in [RSTS11]. Their Velvet system supports language concepts for the definition of multiple independent variability models, which can be composed by inheritance, superimposition, and aggregation. Inheritance allows the extension of one variant by another with additional features and constraints. With superimposition, separate dimensions of one concept can be integrated and, finally, aggregation can be used to combine multiple instances of a variability model (i.e., concrete configurations).

However, with some abstraction, all composition mechanisms for feature models are based on

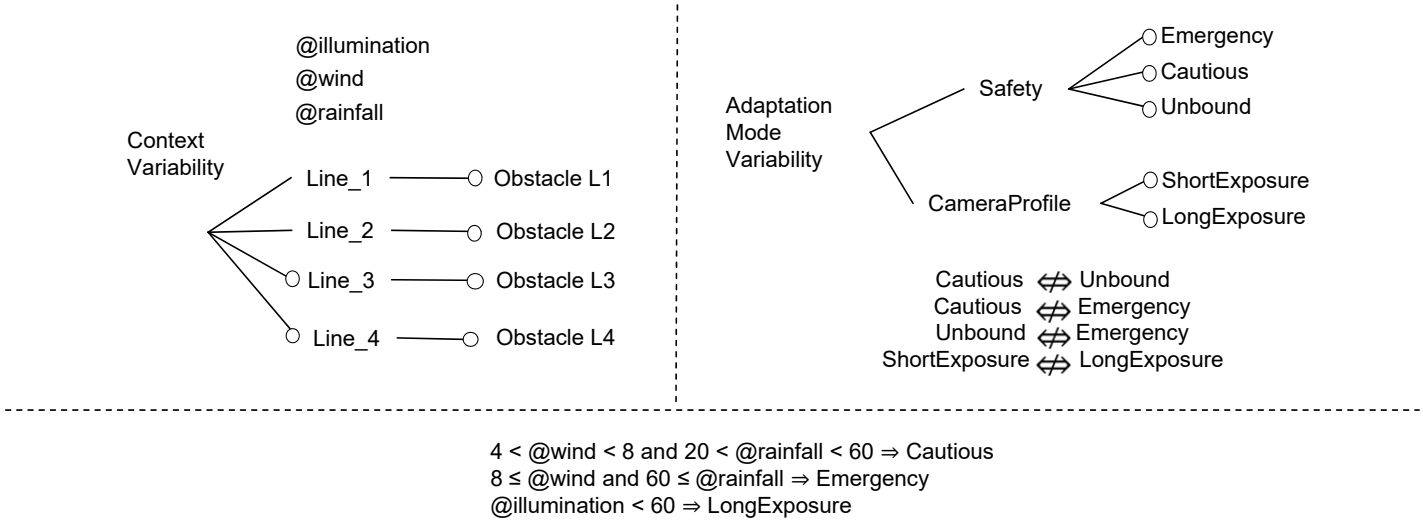


Figure 4.15.: Explicit representation of adaptation modes. *For context and adaptation now separate variability spaces are defined. The adaptation mode variability model describes which adaptations may be configured depending on context situations. Mutual exclusions are modeled as negated equivalences. Both trees are causally connected by cross-tree constraints on the bottom.*

propositional logic as features themselves are considered as symbols without inherent semantics. In contrast to Rosenmüller et al.'s work, testing against interfaces does not require for composition semantics regarding object-orientation. Thus, for the hereby presented concept, it is sufficient to define the causal relation of both context variability and adaption mode variability by propositional logic constraints.

Solution Concept and Example

Employing propositional logic constraints, as used for extending context variability models, is also a reasonable approach for interconnecting context with adaptation modes. The major difference is that now the variability space of the context is separated from a second adaptation-mode-specific variability space. As Figure 4.15 shows, both spaces are again represented by feature trees plus numeric contextual variables. In the newly defined feature tree for adaptation modes, we distinguish between the variability dimension `Safety`, which either may be `Emergency`, `Cautious` or `Normal`, and the `CameraProfile`, which either can be refined to `ShortExposure` or `LongExposure`. For formal simplicity, we model mutual exclusions as negated equivalence conditions.

In a next step, the forward directed causal connection from context variant to adaptation mode can be denoted in the form of implication constraints (Figure 4.15, lower part). Those constraints are based on the semantics we defined beforehand in the definition of context variability models (cf. Section 4.3.4). The first two constraints define in which value ranges we expect the drone two work in `Cautious` or `Emergency` modes, whereas the latter constraint defines under which illumination condition the camera requires the `LongExposure` mode. Using such constraints, the test modeler can restrict the variability space so that only variants are valid, in which context state and adaptation mode are correctly combined.

Definition 4.6 (Joint Context/Adaptation Variability Space): *The major difference to the previously introduced usage of variability definition is that now there may be multiple. To connect both, it can be assumed that the feature spaces are configured together. In consequence, a configuration incorporates the selection of features from both spaces under the consideration of all feature-model-specific constraints and all cross-tree constraints. Formally, this can be defined*

as follows:

- *Given are a context variability model CVM,*
- *an adaptation mode variability model AVM,*
- *and a set of cross-tree formulas Φ_{Cross} over the feature sets of CVM and AVM.*
- *Valid configurations are* $C(CVM, AVM) = C_{CVM+AVM} = \{c_i | c_i \models \Phi_C \cup \Phi_A \cup \Phi_T(CVM) \cup \Phi_T(AVM) \cup \Phi_{Cross}\}$

In summary, all configuration must be valid under all constraints of both trees and all additionally given, connecting conditions.

□

Usage in Test Generation and Simulation

Putting the composite variability space into action only requires to add the adaptation mode model and cross-tree constraints and to consider all together during validation. For this purpose, the `Validator` component now checks the composite variant of context and adaptation mode as a whole. Hence, stimulus models are equipped for reconfiguring the adaptation mode variant synchronously to the context variant. Consequently, the Petri net's application conditions can be defined over both variability spaces.

4.3.9. Context-Dependent Reconfiguration

> Context-dependent reconfiguration addresses the requirement of **expressing stateful adaptation**. With this concept, adaptation cannot only be modeled as direct effect of context situations but also dependent on previous adaptations.

In the previous section, we discussed how context can be mapped to adaptation mode variants of a separate variability space. With stimulus models, contextual change can be defined stateful—a context switch depends on the previous situation. Similarly, adaptation may appear stateful if the determined adaptation mode not only depends on the context variant but the previous mode as well. For instance, if the system specification prescribes that a drone never directly switches from emergency mode to normal mode but instead first goes for a time in cautious mode, there must be a definition of this inter-mode dependency. Another example is if the drone is expected to ignore squall (sharp increases in wind speed) for a while, instead of immediately adapting to it. Similarly, an adaption to decreasing speeds could also be suspended until a specific level of certainty is reached. For this purpose, a history of past wind measurements has to maintained and the resulting function forms a hysteresis. Thus, a mere mapping from context to adaptation mode variant is not sufficient.

Solution Concept and Example

State machines are not only appropriate to express the complex logic of several context dynamics but also for modes of adaptation. Because switching between modes is triggered by contextual change, the description of the change between has to be causally coupled to stimulus models. State machines can communicate by events that are consumed during the execution of a transition (acceptor automaton) or produced (transducer). As the causal connection for adaptation appears unidirectional from context to adaptation mode, events that should be produced have to be defined within stimulus models and be accepted by automata that define the dynamics of adaptation modes. In consequence, the designer of a stimulus model is expected to provide events carrying information on the executed change.

Due to the heterogeneity of stimulus models, the mechanism of event production would be heterogeneous as well. Examples are depicted in Figure 4.16. For instance, the previously used circular graphs could be split in value ranges where each entry in a range produces a specific event, which is illustrated in both Figure 4.16a and 4.16b. Each range is annotated by an event to be produced (indicated by an exclamation mark). In Figure 4.16c, the wind-related state machine is extended with events at each transition. When the context changes, not only the context is altered, but an event is produced as well. Similarly, for movement profiles, in Figure 4.16d, event production can be implemented with spatial zones that throw events when entered. In the illustration, the area layout is separated in a northern and southern zone, whereas, according to variably definition (cf. Figure 4.6), the latter could contain obstacles, which should be reacted to during a drone flight. All those model types are event sources, but implement this feature with different approaches.

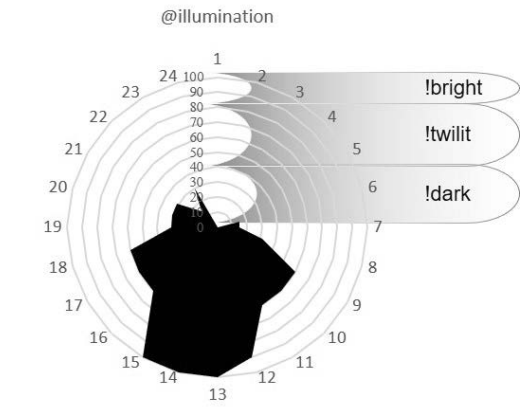
Another requirement to cope with appears when the production of certain events depends on *multiple* contextual aspects. For this purpose, the usage of event processors, which aggregate events and produce derived ones, is required. However, instead of providing a bandwidth of multiple concepts, hereby a single generic one based on a transducer automaton is proposed. The mechanism is presented in Figure 4.17a. The automaton accepts for each transition different events (indicated by a question mark) and produces one (again indicated by an exclamation mark). Only one of the defined input events must occur to trigger the transition. The automaton translates between events and introduces another layer of the state space.

The acceptor automata in Figure 4.17b describe how an SASuT is expected to reconfigure depending on those translated events. Each state represents a certain variant of the adaptation mode variability model. In the following, an automaton with adaptation modes as states is called *reconfiguration automaton*. There are multiple possibilities how configuration can be specified within such automata. Firstly, a single universal automaton could be defined which manages a global adaptation mode. In this case, each state has to relate to a variant which is valid against the adaptation mode variability model. Secondly, single aspects of the adaptation mode may be defined within separate automata. This mechanism follows the same principle as stimulus models, which each is defined for an individual context variability dimension. Hereby, again, transitions may be annotated with reconfiguration actions so that states only implicitly refer to variants, whose validity may be checked at run-time. To reconfigure to valid variants, each feature activation must be processed with the deactivation of potentially mutually excluded features. Thus, the semantics of the entry operations within the automata is that the validity of the reached variant cannot be checked before *all* actions are completely executed. In consequence, these action sequences must be considered as atomic operations. This method gives the modeler more flexibility while, at the same time, additional non-determinism due to potentially concurrently accepted events constitutes. However, for the hereby presented concept, the latter method is proposed to be used because it already has been employed for stimulus models.

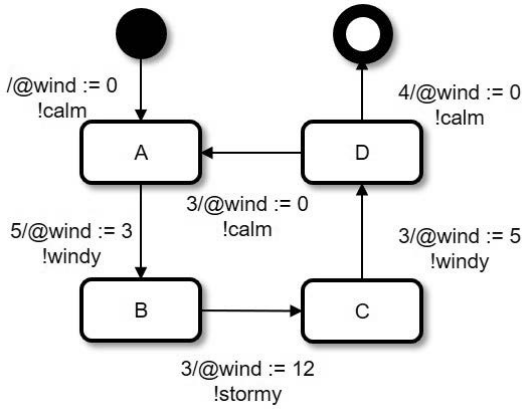
Reconfiguration automata are based on communicating transducers. Hereby, it is beneficial to let designers sometimes mix-up reconfiguration automata with stimulus models. Applications of this concept become relevant when the production of events partly depends on external influence, which would be controlled by timing, and other parts by internal events of the constructed model stack. For instance, the emergency mode of the drone may be triggered by context, whereas an internal condition of its logic could cause it flying home without respecting the inappropriate weather conditions.

Definition 4.7 (Reconfiguration Automaton): *Formally, a reconfiguration automaton can be expressed as a seven-tuple* $(Q, \Sigma, \Gamma, q_0, \delta, \omega, \lambda)$. *Assuming that* Σ_0 *is the set of possible variant manipulations (cf. Definition 4.4), an the components of this structure are the following:*

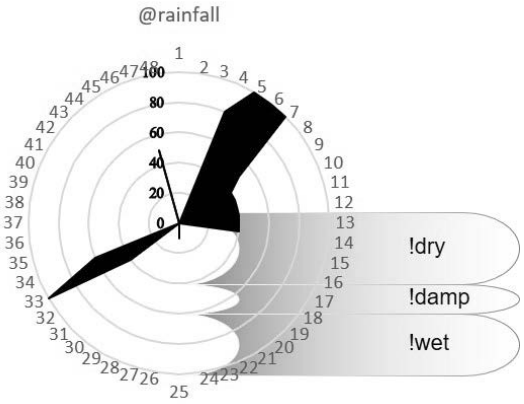
- *A finite, non-empty set of symbolic states* Q,
- *the finite, non-empty input alphabet* Σ,



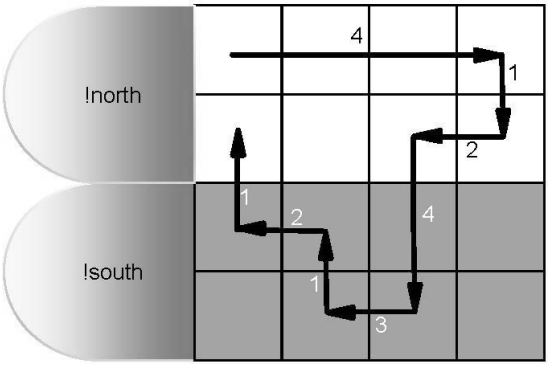
(a) Circular graphs are annotated with value range, whose entry triggers an event for illumination.



(b) Circular graphs are annotated with value range, whose entry triggers an event for rainfall.

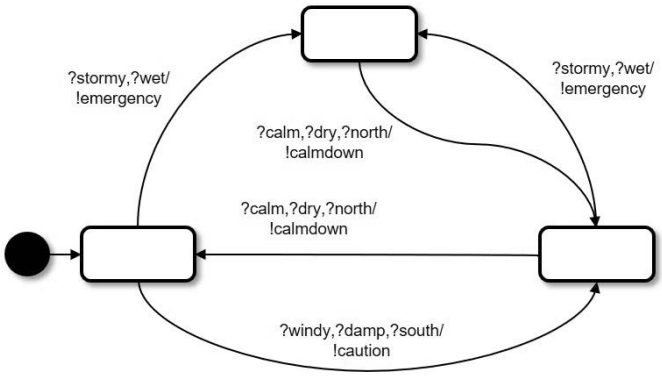


(c) In state machines, event can be appended to the definition of a transition's logic.

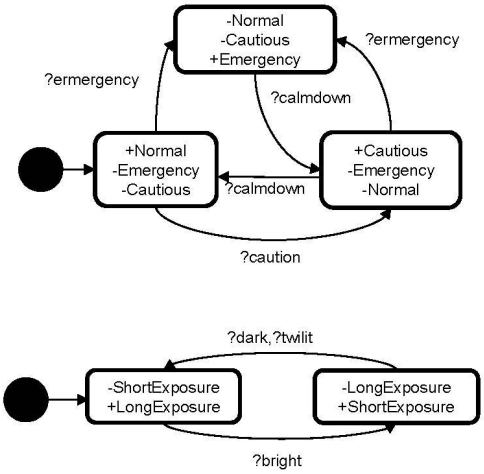


(d) In a spatial layout, spatial zones may be labeled withe events to be produced

Figure 4.16.: Stimulus models with event production.



(a) Transducer automation translating from a set of input events to other events.



(b) Reconfiguration automate control the variant of adaptation mode depending on incoming events.

Figure 4.17.: Causal chain of stimulus and reconfiguration.

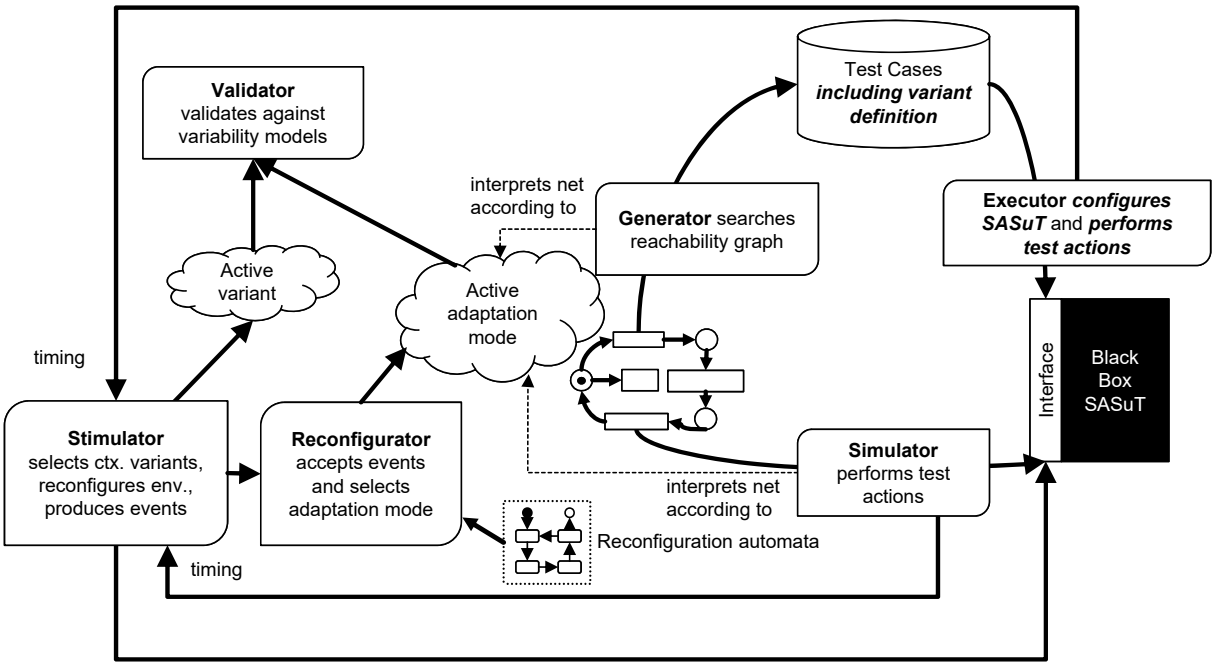


Figure 4.18.: Generation and simulation infrastructure including reconfiguration and explicit adaptation modes. *A separate adaptation mode is maintained and altered by events that flow from context simulation to the* `Reconfigurator`*. The latter takes reconfiguration automata as input.*

- *the finite, non-empty output alphabet* Γ,
- *an initial state* $q_0 \in Q$,
- *a transition function* $\delta : Q \times (\Sigma \cup \mathbb{N}) \to 2^Q$,
- *an output function* $\omega : Q \times \Sigma \times Q \mapsto \Gamma^*$,
- *and a state labeling function* $\lambda : Q \times \Sigma_0^*$, *which maps each state to a word of the action operation alphabet* Σ_0^*.

For the semantics, we assume that all instances of reconfiguration automata are initially in their initial state. With $e_0 \in \Sigma$ *being the first input event in queue (i.e., we assume a first-in-first-out semantic),* q_i *the current state, and* c_{A_i} *the current adaptation mode variant, the set of reachable adaptation modes* C_{A_j} *are all* $c_{A_j} \in C_{A_j}$ *with* $c_{A_i} \to^{a_0} \cdots \to^{a_n} c_{A_j}$ *and* $\lambda(s_j) = a_0 \dots a_n$ *and* $s_j \in \delta(s_i, e_1)$. *The transition algorithm is processed for all inputs in queue as long as events are available.*

□

Usage in Test Generation and Simulation

The decoupling of the variability spaces is visualized in Figure 4.18. Additionally, to the `Stimulator`, a `Reconfigurator` is introduced, which maintains the adaptation mode and its change based on the reconfiguration automata. Both components communicate by an event queue. The variant for context and adaptation mode can be validated separately by the `Validator` and the interpretation of the adaptive Petri net is executed according to the configured adaptation mode variant.

4.4. Adequacy Criteria for SAS Test Models

Although the presented formalization shows that the introduced model types integrate well, finding an adequacy criterion that spans the complete stack is difficult. Using heterogeneous models also leads to using different means for formalizing a criterion that determines when an interpreter, respectively search algorithm, is expected to terminate test-case generation. This challenge especially holds for stimulus models because the introduced interfaces do not fix assumptions on the inner structure of the concrete model type so that defining general criteria is not possible. However, for every single type of model, which was used in this chapter, adequacy was approached in literature:

- *Petri nets:* The single conceptional elements of a Petri net can be used to define adequacy criteria. For instance, in [ZH00], Zhu et al. propose definitions for transition and state coverage, combinations of them, and others based on traces through the reachability graph.
- *Feature models:* Because feature models stem from the body of knowledge of software product lines, relevant criteria should also be adopted from this area. Features are decision points of which elements should be incorporated in a concrete software variant. Consequently, combinatorics of the feature space must be considered by adequacy criteria. For instance, Johansen et al. discuss in [JHF11], how to apply combinatorial testing in the form of a feature-pair-wise criterion. They also discuss that a good design of the feature models is important for later test selection.
- *Finite state machines:* State machines are the most widely-used operational model for test generation. The most common are state, transition, and branch coverage.

All those criteria can be used in combination by defining one for each component of the test model. In case of the finite-capacity labeled Petri net, the restriction of each place element to a maximum number of tokens allows further precise control on the generation. Additionally, defining an overall k-boundedness criterion can limit the Petri net execution to a manageable size of state space.

Furthermore, adequacy criteria interact with the conceptional complexity reduction, which is inherent to presented approach. In the beginning of Section 4.3, we discussed the process of iteratively expanding variability to examine further the SASuT's reacting to context variation. The decision when to terminate this iterative process depends on the remaining state space, which is reachable under the given adequacy criterion. This interaction should consequently be considered when advancing towards more detailed context and adaption mode variability models.

4.5. Discussion on the Viability of the Employed Models

The above-introduced formalisms constitute a model ensemble that has certain qualities, functions, and a purpose. Model theory discusses these and other aspects of models with the goal to evaluate their viability and to compare with alternative approaches for the problems, which had to be solved when designing the models. In [Tha13], Thalheim summarizes the concerns that can be investigated for models in general. For this consideration, a definition of the notion of a model is necessary, of which many exist. However, the most recited one has been proposed by Stachowiak in [Sta73], who lists three important ingredients of models:

1. Mapping property: A model is based on the mapping to a certain origin, which may be a real or virtual object that is represented.
2. Truncation property: The model abstracts from its origin by leaving out details, which are not considered as relevant for the purpose of the model.

	Mapping origin	**Truncations**
Context VM	Contextual situations of the SASuT and their measurable attributes	Only relevant attributes, structural features, classes of values and representatives
Stimulus Model	Behavior of the context and operational logic of environment change	Scenarios are defined without the necessity to describe complete behavioral logic of the context
Adaptation VM	Classes of adaptation (modes)	Classes that symbolize complex adaptation modes
Reconfiguration automata	Operational logic of adaptation modes in consequence of context change	Represent only behavior that is relevant for defined context scenarios
Adaptive Petri nets	Business logic of the SASuT	Considers context change only at certain states (cf. timer transitions), only test-relevant behavior is modeled

Table 4.1.: Model types and their qualities according to the definition of the notions of models by Stachowiak [Sta73].

3. Pragmatic property: Models are not exclusively associated with their origins. Each model has a certain substitution function for users, tools, or a phase of time.

Pragmatism for the introduced model ensemble as a whole is given by the task of creating a test and simulation model for generating adequate test-cases or running discrete ITL simulations. For both the mapping and truncation property, Table 4.1 depicts the respective offerings of the single model types. Each type has a certain origin reaching from contextual situations, their behavior, adaptation modes including operations in the business logic, which is represented by adaptive Petri nets. In all representations, some features of the tested black-box are subject to truncation. For instance, the contextual behavior may only be described in a scenario-like fashion without the necessity to re-create the complete situational state space of reality.

Another considerable aspect that Thalheim emphasizes is the model's *purpose*. To match criteria of purpose, the *impact* of the overall test model is its goal to define the relevant behavioral space of an SASuT. To reach this goal, structural and behavioral concerns of the SAS are modeled so that automated composition, search, and generation can be performed based on the given semantics. Simulation and generation are the specific *function* of the model that is novel in its appearance as a combination of known models and principles from existing research. A crucial *restriction* is that the proposed model is not adequate for tests, where real-time is a major issue.

Furthermore, according to Thalheim, a model cannot exist without a *general model frame*. One pillar of that frame is the founding concepts of the model, including the body of knowledge in SAS engineering, model-based testing, and dynamic variability management (cf. Chapter 2). A second pillar is *structural and behavioral character* of the model, which is, in case of the introduced model ensemble, based on graphs (finite state machines, Petri nets) and decision trees (feature models). Pillar three encompasses the *application domain*, which is the general case of testing an SAS that runs a control loop for monitoring, analyzing, planning and execution. Finally, a fourth pillar is the model's purpose to deal as a *metamodel* for the creation of test model instances.

Finally, Thalheim refers to the notion of the *fitness* of a model, which was originally introduced in [Hal07] by Halloun. The test model for SAS can be said to be fit if it is (a) useful, which we have support for as it allows for generation of test-cases and ITL simulation, (b) has potential to fulfil its purpose (models are executable and precise enough for the description of discretly-timed test-cases), (c) if the model is efficient (there are no extra parts, which define more than necessary for generation and simulation), and, finally, (d) if it does not lack generality, which is the case because a test model of the proposed appearance can be employed for SAS without being restricted to a certain architectural singularity from real-world implementations.

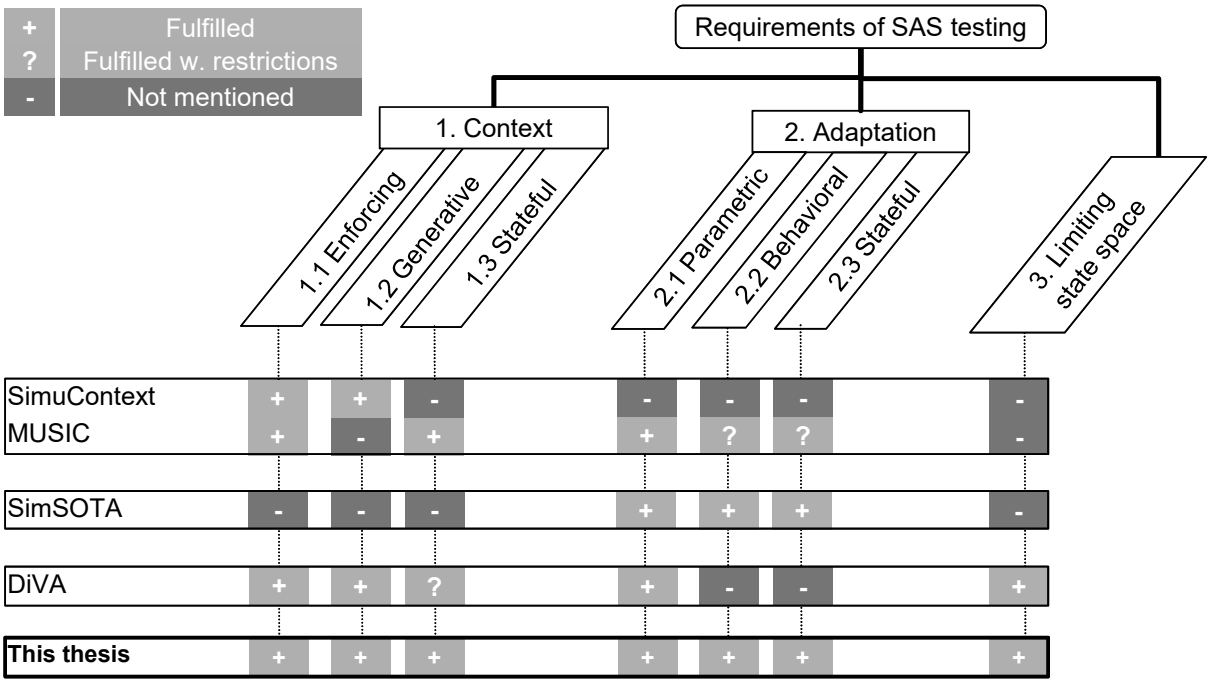


Figure 4.19.: Comparison of the proposed approach to related work.

In summary, the investigation of the discussed model properties shows that the artifacts that are targeted to be used for test modeling SAS are indeed viable for their purpose. However, another question is how they relate to existing work, which is the subject of the next section.

4.6. Comparison to Related Work

The concepts proposed in this thesis have been designed to provide a full-fledged and integrated solution for model-based testing of SAS. Thus, all requirements, which were explored in the previous part are covered as Figure 4.19 suggests.

In comparison to the test approaches in the domain of context-aware computing, the concepts that are proposed in this thesis allow for enforcing pre-designed contexts in a stateful manner. For this purpose, we use context variability models, from which context-dependent test-cases are generated, which is the central capability that is required to cope with the potentially enormous number of relevant states to be covered. Also, in contrast to DiVA, the stateful generation can be controlled by the use of stimulus models.

On the adaption side, the proposed concepts' bandwidth is comparable with SimSOTA and, thus, outperforms the remaining ones. Parametric adaptation, which is expected under a certain context situation, can be defined in the adaptation mode variability models, adaptive behavior in adaptive Petri nets and reconfiguration automata allow for a describing stateful adaptivity. A further similarity to SimSOTA is the ability to compose certain aspects of adaptation. Whereas SimSOTA tackles this challenge by hierarchy, stigmergy, and direct interaction of explicitly modeled control loops, the proposal of this thesis is to wire different reconfiguration automata by event flow.

Finally, state space limitation is supported by the ability to define variability constraints and especially timer transitions in adaptive Petri nets as restriction of when context changes are tested in relation to service behavior. Additionally, adaptive Petri nets provide boundedness constraints as a useful control mechanism for state space exploration.

4.7. Summary and Discussion

In this chapter, an integrated modeling approach has been presented, which allows for specifying test models for self-adaptive systems. Each type of model has a formalized syntax and semantics so that their exact interaction can be understood. Also, the data flow and processing in the case of test-case generation or ITL simulation has been discussed step-wise with each newly introduced concept. A tester can make use of these concepts up to the level, which is appropriate for his or her required software quality and the maturity of the concrete SASuT.

Starting with a Petri-net-based representation, even a system with parallel processes can be test-modeled. The global context, whose variability is represented as feature model, connects all parts of a potentially distributed system, which also can be a restriction of the overall modeling power in this approach. Context variation is modeled in stimulus models, which can be defined freely as long as the concrete models adhere to the introduced stimulus model interface. By using the concepts adopted from FPN, we define behavioral adaption depending on the selected context variant. Stimulus models manipulate this current variant and are operated from budgets of virtual time, which is produced from an external clock or, in an advanced approach, from timer transitions within Petri nets. Additionally, stimulus models can be equipped with more powerful operation logic by wiring them via the exchange of events.

In summary, the presented concepts are designed to match the requirements, which were introduced in the previous chapter and, thus, allow for investigating much deeper the correctness of expectations to self-adaptive systems than the approaches presented in related work.

5. Model-based Adaptivity Test Environment

This chapter is based on the following publications:

- Georg Püschel, *Test Modeling of Dynamic Variable Systems Using Feature Petri Nets.* Technische Universität Dresden, Fakultät Informatik. ISSN 1430-211X, TUD-Fl13-01-Sept. 2013. Technical Report, 2013.
- Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Thomas Schlegel, Uwe Aßmann, *A Combined Simulation and Test Case Generation Strategy for Self-Adaptive Systems.* Journal On Advances in Software, 7(3&4), pp. 686-âĂŞ696, 2014.
- Georg Püschel, Christian Piechnick, Uwe Aßmann, *Generative und simulative Softwaretests für selbstadaptive, cyber-physikalische Systeme.* In Proceedings of the Multiconference Software Engineering and Management 2015, Koellen-Verlag, 2015.

The applicability of the models and methods that were proposed in the previous chapters is demonstrated in the following in the form a reference implementation named *Model-Based Adaptivity Test Environment* (MATE). MATE aims at making test engineers capable of performing a complete dynamic test of self-adaptive systems, which incorporates test design, environment setup, test execution, and reporting.

Convenient test environments support engineers with all these tasks by providing an appropriate and homogeneous toolset. However, to implement the introduced SAS test concepts, MATE focuses on creating and executing the respective test models. As discussed in Section 2.2, in MBT, test models substitute manual test specifications. In consequence, the test design and implementation process are of special importance within the MATE toolchain.

For executing the modeled actions within a given technological space, MATE also requires components for creating test drivers. These components are called test-automation connectors in MATE. The test environment provides a framework to create these connectors and supports engineers with attaching MATE to an SUT.

In test-execution, we distinguish between running generated test-cases and interpreting the test models directly within a simulation while communicating with a connected environment in the loop. For both approaches, MATE is equipped with a model interpreter and algorithms for searching the state space. Both test generation and simulation have to be able to make use of the test-automation connectors. While execution, MATE is expected to record the results in an appropriate report format, which contains information for reflecting which expectations failed

under which conditions. Within such a report, incidents are reported as intended by the standard test process.

This chapter contributes the following proposals coping with the SAS test challenges:

- A reference architecture for an **integrated test environment** (MATE) that realizes the proposed models and allows for employing them along a standard dynamic test process
- A tooling environment, which enables testers to perform all tasks of a standard process of dynamic testing based on the introduced models and architecture
- Outgoing from the mathematical formalization in Chapter 4, a re-formalization as object-oriented metamodels
- A flexible operator-based framework for extending the models or specifying new stimulus models.
- A test-automation framework for SAS testing
- An interactive user interface for step-wise simulation of SAS

In the following, these contributions are presented in detail. In Section 5.1, the technological foundation of MATE is discussed. In Section 5.2, the overall structure of the complete test framework is outlined. Afterwards, Section 5.3 presents the implementation of the conceptional model within MATE. The next Section 5.4 describes how a generator framework implements the semantics of these models. Section 5.5 discusses the test-automation framework for building technology adapters. Afterwards, in Section 5.6, the top layer consisting of user interface and tooling is shown, which support users in complying with the standard process of testing.

5.1. Technological Foundation

This chapter shall define a reference implementation and reference architecture of the presented approach so that a concrete technological foundation had to be found. However, whereas the details of the hereby described realization may be special to the chosen technology, in principle, all given models can be translated to arbitrary object-oriented platforms as well as algorithms. For the hereby introduced solution, we chose Java as representative of this category of possible technology spaces. Also, editors and newly defined textual languages do not depend on a concrete base technology but only on the availability of a parser framework.

All components of MATE are implemented based on the Eclipse Rich Client Platform[1] (RCP). RCP has been the basis for different integrated development environments (IDEs) with a special focus on Java. The components that are provided by Eclipse are called plug-ins and are deployed as bundles on Equinox, an implementation of the Open Services Gateway Initiative (OSGI) standard[2]. Equinox allows for dynamically loading bundles (i.e., at run-time and on demand) and resolving dependencies. Graphical user interfaces are built based on the Standard Widget Toolkit (SWT), which is an essential part of RCP instances.

Additionally, the Eclipse Modeling Framework (EMF) includes Ecore, an implementation of OMG's Essential Meta Object Facility, a metamodel together with sophisticated tooling, which includes textual editors and code generators. These features allow developers for creating their domain-specific metamodels and generating Java code and editors from them. For MATE, this capability is used for realizing its SAS test metamodel

[1] `http://www.eclipse.org`
[2] `http://www.osgi.org`

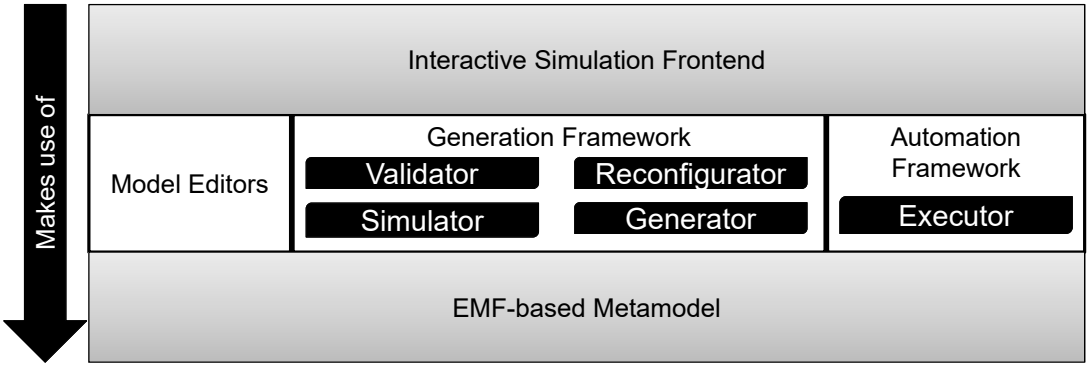


Figure 5.1.: Layers and components of MATE. *Based on EMF, the MATE metamodel implements the introduced formalisms. The generator framework contains the metamodels' semantics and a set of algorithms for exploring the models' state space. Consequently, the generation component also incorporates all functionality of the context validator, reconfigurator, simulator, and generator, which were introduced in the previous chapter. Furthermore, the test-automation framework allows for building technology bridges, which map test actions to code. The resulting adapters are test executors, which can be reused for the given technology space. Testers are equipped with an interactive simulation frontend to control and observe the simulation and test-automation process step by step.*

Graphical models are created in editors based on the Graphical Editing Framework (GEF), another project of the Eclipse community. The framework contains an API and several abstractions that help to map EMF models to graphical entities and vice versa.

Besides graphical representations, the introduced models incorporate several textual notations. To parse these notations, the jparsec framework is used. jparsec[3] is a parser combinator framework and allows for building lexers and parsers directly from a Java-based configuration even at runtime.

Another technology that is used by MATE is Sat4J[4], a Java reasoning framework. This component is required to solve propositional logic and constraints within the introduced models.

Finally, MATE provides extension points, which can be filled with scripts that are programmed with the Scala[5] language. Scala is an object-oriented, functional language, which directly compiles to Java bytecode and, thus, can be executed directly in a Java runtime environment. This feature supports the test modeler with co-evolving test drivers with the model and introducing more complex functions that are evaluated during simulation or test-case generation.

All these third-party technologies provide the lowest layer of MATE and are instrumented by it. The next section outlines MATE's coarse-grained layers that realize the announced contributions.

5.2. MATE Base Components

MATE organizes its components in layers. This overall hierarchy is depicted in Figure 5.1. Black boxes denote where the conceptional components from the previous chapter are realized. The metamodel is the foundation of all other framework parts. Engineers design and edit instances of the metamodel by the use of the graphical textual model editors and test-cases, which can be generated with a generator framework.

The generation framework provides an implementation of the metamodel's semantics. Respectively, it comprises components that perform the interpretation of the adaptive Petri net,

[3] `https://github.com/jparsec/jparsec`
[4] `http://www.sat4j.org/`
[5] `https://www.scala-lang.org/`

the simulation of stimulus models, and the reconfiguration driving by reconfiguration automata. Furthermore, the generator can reason about the current state and its validity to model-inherent constraints. Thus, propositional logic, which is derived from feature trees, can be checked as well as satisfiability problems from custom stimulus models if required.

The generation framework supports the production of test-cases using a traversal strategy, which is specified within the toolchain. These test-cases comprise executable sequences of actions, which are stored as instances of another metamodel. To support SAS test-case execution, MATE provides a test-automation framework, which consists of an API and templates. Both support building test executors that adapt the model for certain technology spaces.

On top of MATE, a simulation frontend is introduced to execute test models in the loop. Testers can directly interact with it so that they are able to intervene the models' interpretation and steer test-automation step by step. Alternatively, the simulator can perform the execution automatically (depending on whether test actions are completely automated) and visualize the state of execution within the model editors. Both executed test-cases and simulation result in test reports, which are also based on the metamodel.

The framework is designed to be extensible and reusable. This means that new metamodel elements, including syntax and semantics as well as additional target systems, can be adopted with low effort and all included elements are generic in the sense of the domain of the SASuT. The single components of MATE are described in detail in the following sections.

5.3. Metamodel Implementation

MATE's foundation is a strictly formalized metamodel, which implements the introduced formalisms, including graphical and textual notations. Graphical representations are feature models (cf. Section 4.3.4), adaptive Petri nets (cf. Sections 4.3.3, 4.3.5, and 4.3.7), stimulus models (cf. Section 4.3.6), and reconfiguration automata (cf. Section 4.3.9). Additionally, we distinguish four components for the textual representations:

- **Variability constraints** model application conditions for FPN transitions, which are logic propositions on the selected variant.
- **Functions** extend variability constraints so that certain propositions can be externalized from the graphical models.
- **Terms** are abstract syntax trees (ASTs) for representing expressions in form of C-like function calls. These expressions are used as a flexible format for transition labels to be interpreted by the test-automation connector.
- **Test actions** are commands, which are written on Petri net labels and are used for modeling the change of environment conditions or configurations as well as verifications. In contrast to term-based actions, they have special semantics, which is implemented in MATE and not in the test-automation connector.

Furthermore, metamodels for test-suites and test reports are added. All these formalizations are based on EMF/Ecore, which allows for homogeneously processing all artifacts.

Textual notations are reused in multiple representations so that several model dependencies are established. These relations are illustrated in Figure 5.2. Each model part is implemented in an EMF package, which is denoted in brackets in each of the elements. Adaptive Petri nets, stimulus models, and reconfiguration automata use textual constraints and functions to define conditions as well as actions and terms to enforce a certain state. All definitions of conditions and variability mechanisms depend on the current feature-based variant. Furthermore, test-suites consist of test-cases, which are sequences of steps. A step may incorporate a certain action. An

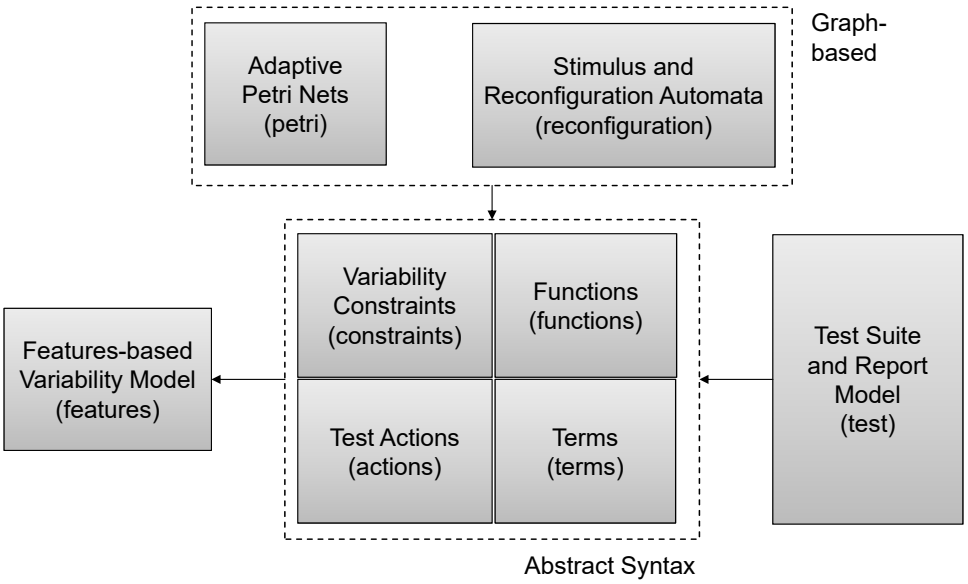


Figure 5.2.: Packages of the MATE metamodel with dependencies. *Variability and configurations are defined within the feature-based variability model. For all textual notations (constraints, functions, terms, and actions), parsing relies on a metamodel of abstract syntax, which depends on definitions of features and attributes. Petri nets, stimulus, and reconfiguration models incorporate elements, which make use of textual language elements. The same holds for test-suites and reports, both containing test steps and, respectively, actions.*

action is modeled within a respective abstract syntax. In the same manner, reports contain actions to document which failures have been observed after certain test steps.

Each part of the presented metamodel is directly related to the concepts and models presented in the previous chapter. In the following, the models are leveraged to a concrete implementation and discussed in detail.

5.3.1. Feature-based Variability Model

In the proposed test metamodel, abstract variability (i.e., context variability and adaptation modes) is defined as feature tree. Figure 5.3 depicts the structure of this metamodel. A `Feature` has a unique name and represents a general standalone variability abstraction, whereas the metamodel also introduces `TreeFeature`s that is specializes to a tree-shaped structure. Within a `FeatureTree`, instances of `TreeFeature` span a hierarchical structure, whereas a feature can be mandatory or not and may have several child features. The `FeatureTree` deals as a top-level entity to make the complete structure referable.

Usually, in testing, equivalence classes of input or output data are specified. Each of those classes may be associated with one or multiple representatives. Thus, testers avoid combining all possible values of the relevant equivalence classes and minimize effort. To support the design of such equivalence structures and representatives, the type `DataClass` is provided, which can contain children that are instances of literal elements of type `DataValue` or `NumergicDataElements`. Both classes are specializations of `DataElement`, which implements the composite pattern [GHJV94]. All elements of this type constitute a finite value space. Classes for numeric elements of super-type `NumericDataElement` span a similar composite structure, incorporating the types `NumericValue` and a `NumericRange`. The latter saves specification effort and is automatically expanded at deployment time to single values.

A `Configuration` that builds up a concrete technical setup, context situation, or adaptation mode is composed of features and `DataBindings` that assign `DataElements` to `DataValues`.

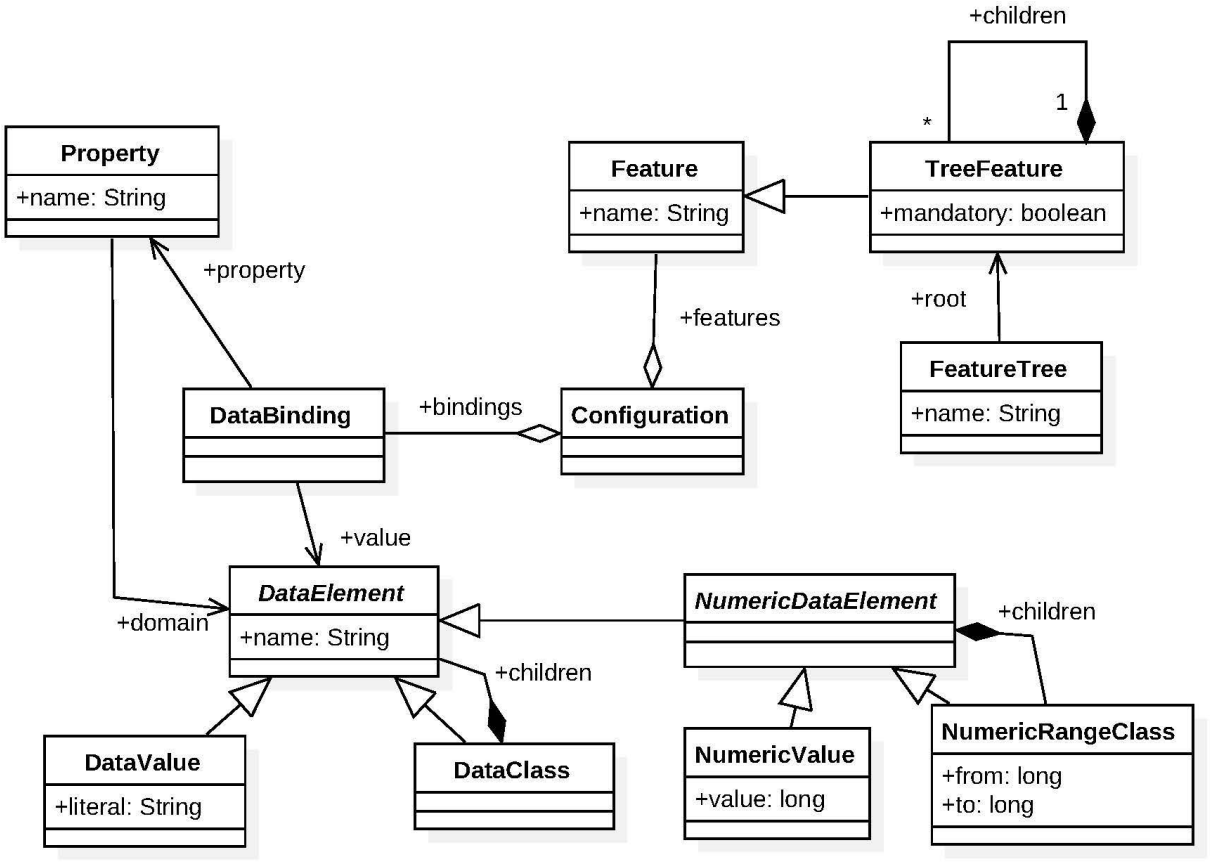


Figure 5.3.: Feature-based variability metamodel (package `features`). *Features can be defined standalone or in trees; properties are defined within finite domains. Selections of features and assignments of property values constitute configurations.*

The feature-based variability model implements all concepts as proposed in Section 4.3.4 and adds several convenience elements, such as data ranges. Based on this model, the test modeler is enabled to specify the complete configuration space as well as individual configurations of the context situation and adaptation model variability.

5.3.2. Abstract and Concrete Syntax for Textual Notations

Graphical representations for the proposed metamodel can be supported by textual notations, which require an abstract syntax. As briefly introduced in Section 5.3, syntax shall be provided for variability constraints, functions, terms, and test actions. In the following, the syntax of these notations is introduced and elaborated by example.

Variability Constraints

A `FeatureAtom` evaluates to true if the referenced feature is selected by the current configuration. Similarly, `DataAtom` evaluates to true if the referenced `DataLeaf` is bound to the referenced `DataElement` by a configured `DataBinding`. Both types of atoms suffice to represent simple application conditions. For instance, the following listing shows some valid expressions:

```
1 Obstacle_L1 and wind = 60
2 (Obstacle_L4 or Obstacle_L2) and not illumination = 0
3 [illumination > 0] or Obstacle_L3
```

Listing 5.1: Examples of application conditions.

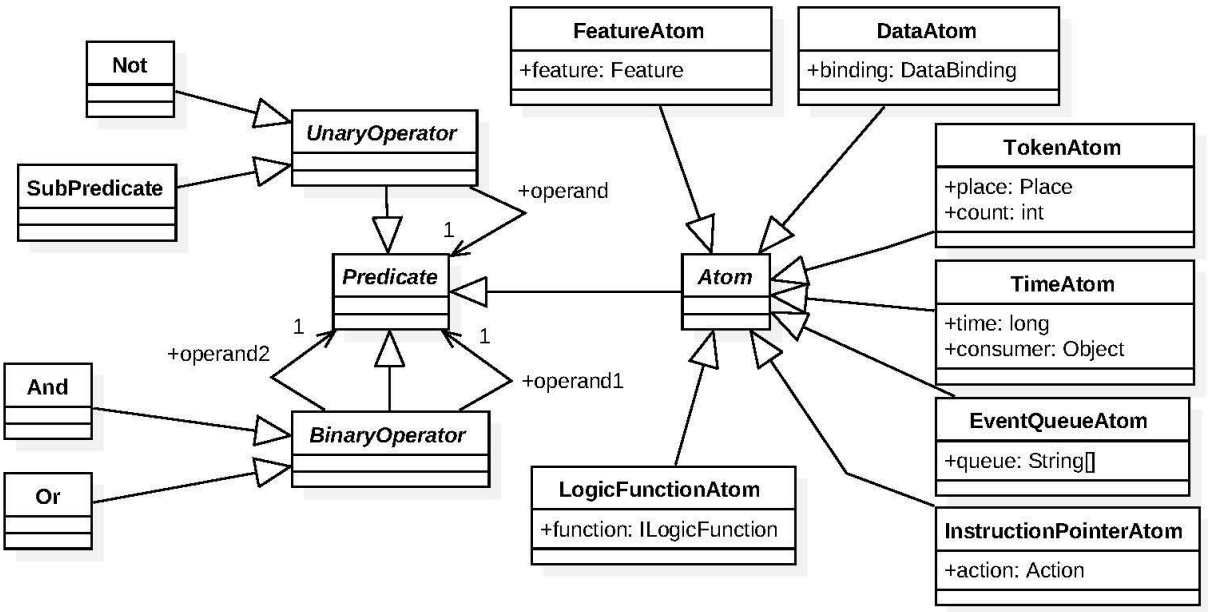


Figure 5.4.: Classes and relations of the abstract syntax for variability constraints (package `constraints`). *Different predicates can be defined: atoms on data assignment and feature selection. Unary and binary operators enable to build complex logic expressions. During interpretation, the state is stored in form of atoms specifying tokens, time, events, instructions, and evaluation results of logic functions.*

Whereas the first expression conjuncts a feature atom and a data atom, the second line illustrates the use of bracket expressions and logic negation. On the third line, the left operand is an instance of logic function atom, which is written in square brackets.

Despite the constraint metamodel being an abstract syntax, it includes several additional atoms for representing the state of interpretation. For these cases, no concrete syntax is needed because the respective objects are not intended to be represented visually. For instance, the `TokenAtom` represents the marking of a certain Petri net place with a given number of tokens. Furthermore, the `TimeAtom` represents the amount of virtual time, which was produced by timer transitions, and can be consumed by a specific stimulus model referred to by the `consumer` attribute. In the same manner, the current event queue state is saved as `EventQueueAtom`. The next action to be executed with a transition label is indicated by an instance of `InstructionPointerAtom`. Finally, a `LogicFunctionAtom`s can contain a logic function (cf. next section) and stores whether the function evaluated to true in the respective interpretation step. All these logic primitives permit a unified maintenance of state during generation or simulation and, thus, ease the extension of the model for custom concepts.

Functions

During the generation of test-cases or simulation, the state of the interpreter represents knowledge that can be used to check predicates or to produce test steps according to the current state. Besides explicit information conveyed by instances of `Atom`, a deduction mechanism over logic functions can be introduced to externalize knowledge from the graphic notations. This mechanism makes knowledge available that is computed at run-time and, afterward, can also be subject to constraint checking and control flow determination.

Logical functions are specified by instances of `ILogicFunction` as presented in Figure 5.5. Several types of logic functions exist. The `ArithmetricLogicBinaryFunction` checks for equality (`EqualFunction`) or inequality (`LargerThanFunction`, `LowerThanFunction`) of two numeric values. These values can be computed from instances of `IArithmeticFunction`, which are binary

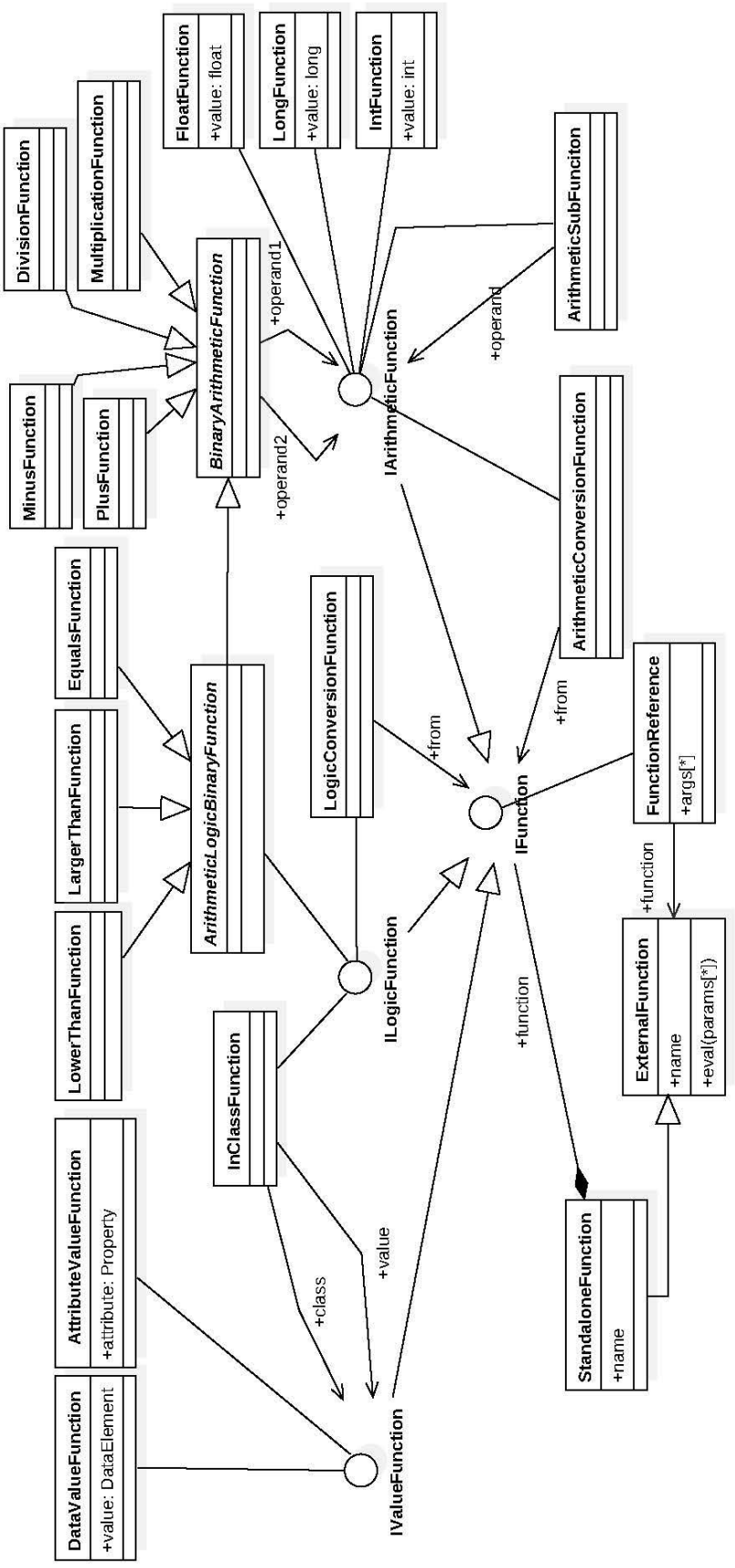


Figure 5.5.: Abstract syntax of functions language (package `functions`). *Logic functions evaluate equality or inequality of numeric function results or literal data values. Both are automatically transformed into each other.*

(BinaryFunctions as division, minus, plus, and multiplication) ore primitive (i.e., non-ary) for the definition of float, int, or long values. Furthermore, the type ArithmeticSubFunction for the definition of bracket expressions exists.

A certain DataElement can be referenced by DataValueFunction and the assigned value of an attribute is accessible by AttributeValueFunction. As data classes contain literal values (DataValue), further functions for their comparison are introduced as sub-classes of DataValueFunction. For the purpose of equivalence class testing, a special class InClassFunction is introduced. It evaluates to true if a literal or typed data element was defined in the sub-tree of a given equivalence class (DataClass).

All different types of functions inherit from the IFunction interface. Based on LogicConversationFunction logic results from arithmetic evaluation (zero is mapped to false, all other to true) and data values (by their name or numeric value) can be mapped to truth values. Similarly, ArithmeticConversation specifies a mapping to the numeric space. Non-successful mappings lead to run-time errors, which the modeler must handle.

The following listing contains two valid definitions of standalone functions. The first is parsed to a LargerThenFunction with operands of type AttributeValueFunction and PlusFunction. The second line represents an example of the InClassFunction.

```
1 function foo = illumination > (60 + wind)
2 function bar = illumination in L
```

Listing 5.2: Examples of function to be used in MATE.

In MATE, functions are tools to compute logic propositions from information that is available at run-time. They can be part of a constraint about a feature tree or may be used in application conditions in adaptive Petri nets. For this purpose functions can be wrapped as a predicate (cf. LogicFunctionAtom in Figure 5.4). Functions may not only occur in a single transition's application condition but can also be reused among multiple transitions (c.f. Section 4.3.5). The classes FunctionReference and StandaloneFunction implement this principle. The first references the latter by its name. A standalone function is a container for an arbitrary function and allows for externalization of its textual specification. The following listing illustrates function usage. Whereas the first line defines a function with parameters, in the second one makes use of the same by calling it with respective arguments.

```
1 function foo(x) = illumination > x
2 function bar = &foo(60)
```

Listing 5.3: Examples of parameterized functions.

However, the functional language provided with the introduced abstract syntax is by far not general purpose. Consequently, a Scala-based interfaced shall be provided to allow for defining functions, which can be called from within a model instance. An example is the following Scala listing:

```
1 def MATE_foo(ctx:MATEContext,rs:ResourceSet,x) =
    ctx.getValue("illumination") > x;
```

Listing 5.4: Example of a Scala-based external function.

Each Scala-defined function must start with the MATE_ prefix to be recognized. The code is compiled to Java class files and automatically recognized by the MATE tooling, which creates respective model elements of type ExternalFunction. The MATEContext ctx, which is provided as first the parameter, allows for accessing the current interpretation state, whereas the ResourceSet rs allows for introspecting the EMF model structure. In this way, the Scala-based evaluation provides great expressiveness in application conditions and feature constraints.

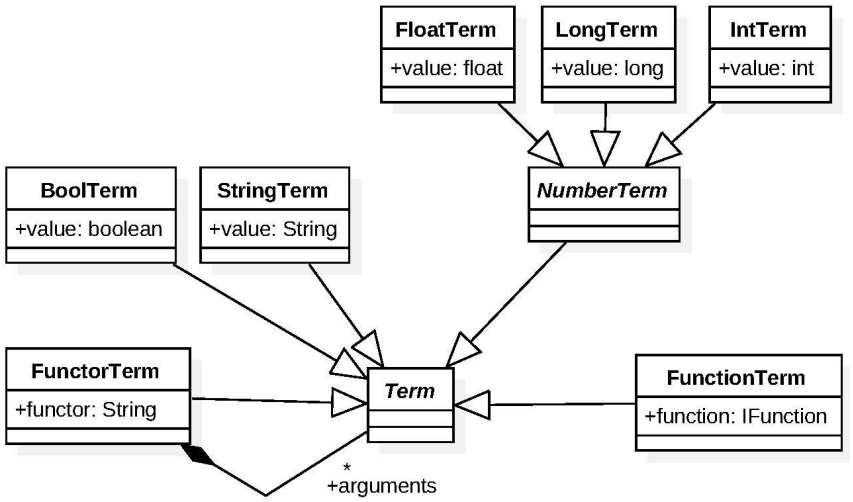


Figure 5.6.: The abstract syntax of the term language (package `terms`). *Besides primitives like strings, booleans, and numbers, complex terms like lists, and function calls with parameters can be expressed. Terms containing functions allow for defining templates, which are evaluated at execution time.*

Terms

The term sub-language allows for expressing C-like operation calls, which deal for a universal notation to be interpreted during test-automation. Figure 5.6 depicts all classes that are necessary to built such expressions. The most general class is `Term`, which generalizes primitive and complex term types. Primitives are `BoolTerm` for boolean expressions (`true`, `false`), `StringTerm` for strings, and `NumberTerm` for expressing float, integer, and long numeric values.

`FunctorTerm` allows for listing parameters and have a `functor` name so that a function or procedure call is expressed. In this manner, tree-structures (i.e., composites) are established. The following listing shows some valid term expressions:

```
1 true
2 42
3 "foo␣bar"
4 foo(bar,42,"foobar")
5 bar(&foo)
```

Listing 5.5: Examples of different types of terms to be used in MATE.

Lines 1 to 3 illustrate the use of primitive terms, whereas Line 4 illustrates to use of a composite `FunctorTerm`. Finally, in Line 5, a `FunctionTerm` is wrapped by another functor term. The concrete syntax needs to escape from the term language by the use of escape sign `&`.

Function terms are part of the introduced variability mechanisms that available in adaptive Petri nets. In the next section, actions are introduced which encapsulate terms or manipulate the selected variant.

Test Actions

Term-based actions are not the only type of command that can be executed by transitions and other model elements. Especially in context stimulation models, attribute values and selected features are altered, which is implemented in a single abstract metamodel, which is presented in Figure 5.7.

The most general model element is `Action`. We distinguish between actions that the modeler writes in the models and the generator or simulator interprets, and `PostGenerationAc-`

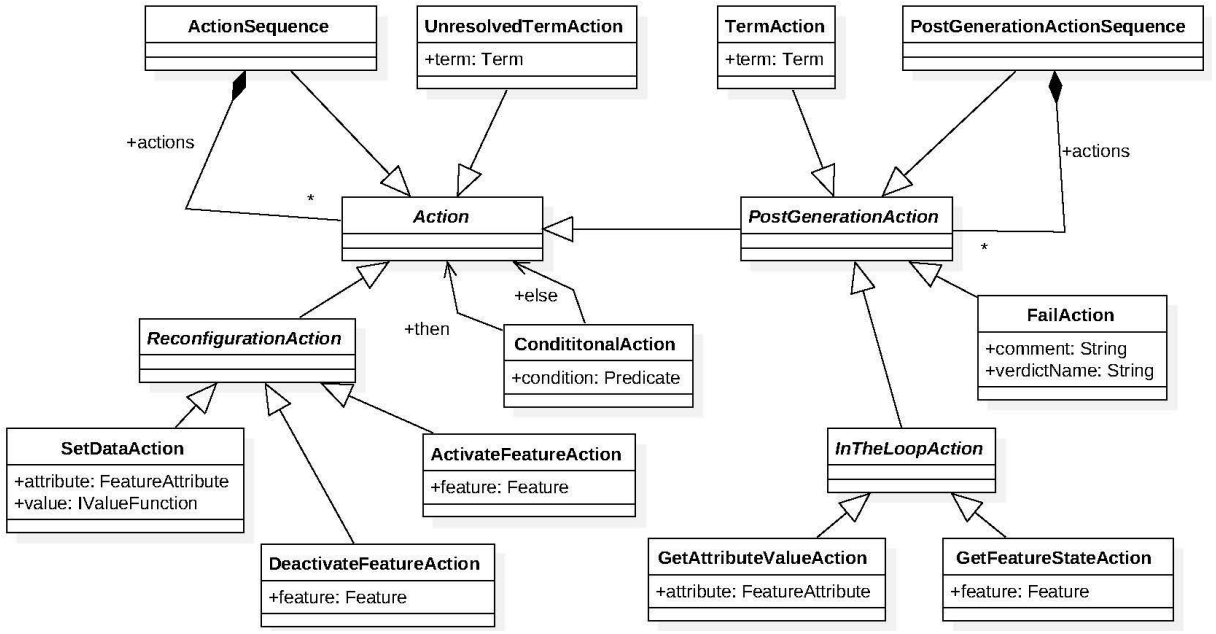


Figure 5.7.: The abstract syntax of the test action language (package `actions`). *Pre-generation actions are specified within a textual notation and are translated to post-generation actions, which involves reconfiguration or template execution with terms. Dependent actions are computed on the basis of sensed input values from the in-the-loop interface.*

`tion`s, which are ready for test-automation. All instances, which do not implement the interface `PostGenerationAction`, have to be transformed to the latter type before test-automation. `ActionSequence` and `PostGenerationActionSequence` built blocks of actions, which are executed sequentially.

The reason for a transformation can be either the necessity to execute a certain variability mechanism or to run a reconfiguration. In this sense, the `UnresolvedTermAction` contains a `Term` which may contain a function, whose evaluation result is embedded as a template expression. The interpreter evaluates at run-time encapsulated term templates and creates a completely resolved `TermAction` that can be applied against the SASuT.

`ReconfigurationAction`s reconfigure the context or adaptation mode variant by activating (`ActivateFeatureAction`) or deactivating (`DeactivateFeatureAction`) features, or by setting data value assignments (`SetDataAction`). For reconfiguration actions, no transformation targets exist because they do not affect the real test object.

Furthermore, several instances of `InTheLoopAction` can be specified. Both sub-classes `GetAttributeValueAction` and `GetFeatureStateAction` enable the interpreter to change the simulation state depending on the conditions sensed from physical conditions. This mechanism allows for implementing model-in-the-loop testing.

An instance of `FailAction` enforces an abortion of the simulation with verdict FAIL, which a tester might specify within a transition to denote a point in the model's state space that should never be reached by the SASuT.

The following listing illustrates the concrete syntax of actions:

```
1 do(sth);
2 +Obstacle_L4;-Obstacle_L3;
3 wind := L;
4 &get illumination;&get value wind;
5 &fail;
```

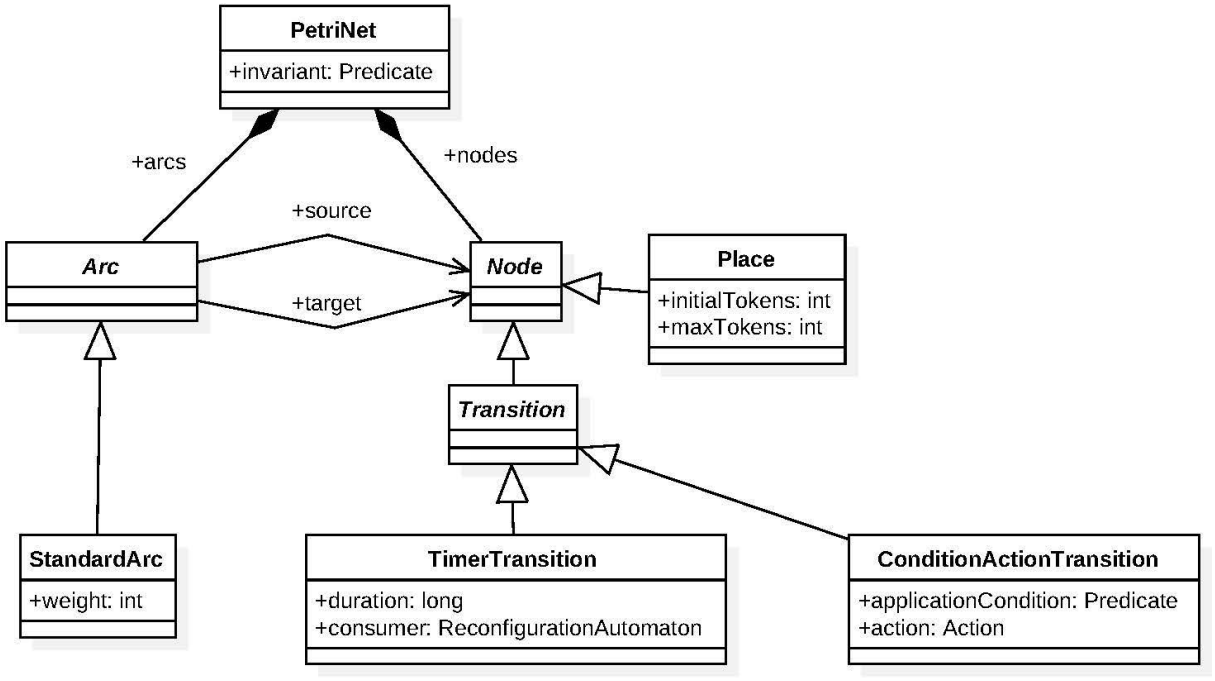


Figure 5.8.: Metamodel for adaptive Petri nets (package `petri`). *Adaptive Petri nets comprise arcs with weights and nodes of different sub-types. The latter can be places or transitions, whereas condition action transitions implement FPN semantics. Additionally, timer transitions comprise a duration and a consumer to be steered from the generated time portions.*

Listing 5.6: Code example for using different types of test actions in MATE.

The parser creates a single `ActionSequence` from this code. In Line 1, a `TermAction` list denoted. Lines 2 and 3 list reconfiguration actions. Line 4 contains a `GetFeatureStateAction` and a `GetAttributeValueAction`. Finally, in Line 5, the use of `FailActions` is shown.

Modelers can employ the action language in stimulus automata for manipulating the selected variant or within Petri net labels. Th graphical representations of the test metamodel are defined in the next sections.

5.3.3. Adaptive Petri Nets

The metamodel of adaptive Petri nets offers syntactical elements for implementing FPN semantics and generating virtual time that further steers stimulus models. Figure 5.8 depicts all necessary model elements. Instances of `PetriNet` aggregate `Arcs` and `Nodes`. Additionally, an `invariant` of the type `Predicate` (from package `constraints`) can be defined, which allows for exactly specifying the variability constraint that has to be adhered to at all times.

An arc has a `source` and a `target`. The types of both must be different (i.e., either `Transition` or `Place`), which is specified by an additional constraint. The `StandardArc` has a weight for specifying how many tokens are consumed or produced during the execution of a target transition or, respectively, source transition.

A `Place` has an attribute to define the number of initial tokens. Additionally, a value `maxTokens` can be defined, which reflects the boundedness constraint of a place. A transition generalizes two different classes. First, the `ConditionActionTransition` implements the FPN-specific application condition as `Predicate` and the label as `Action`. Second, the class `TimerTransition` introduces the syntax that is necessary for steering stimulus models from the adaptive Petri

net. Hereby, a `duration` is specified. At execution time, this duration is added to the values of `TimerAtoms` (cf. Figure 5.4) and later consumed by the stimulus models. The latter are described in the next section.

5.3.4. Stimulus and Reconfiguration Automata

The structures and representations of potential stimulus models are manifold, as we only defined an interface, which all of them have to adhere to, in Section 4.3.6. The restrictions of this interface prescribe that stimulus models define change controlled by portions of virtual time, use reconfiguration actions, and produce events to be consumed by reconfiguration automata.

However, MATE provides a basic implementation of automaton-based stimulus. The only difference of such a stimulus automaton to a reconfiguration automaton is that the first one produces events, whereas the second one consumes them. In consequence, it is efficient to specify their syntactical elements within common metamodel, which is presented in Figure 5.9.

A `ReconfigurationAutomaton` comprises several states and a start state to indicate the initial condition. Furthermore, instances of type `ReconfigurationArc` relate always states. Arcs are conditioned and can specify an action to be performed.

For stimulus models, `TimedArc`s are used; they define a `duration` how much portions of time are consumed. Otherwise, for reconfiguration automata, the `EventArc` class enables to specify an event name, which triggers the transition.

5.3.5. Test Suite and Report Model

In MBT, it is appropriate to store the result of test-case generation and test-case execution as instances of a metamodel to foster a more convenient test-automation or, respectively, evaluation. The structure of MATE's test metamodel is illustrated in Figure 5.10.

`TestCase`s are a sequences of `TestStep`s and aggregated by a `TestSuite`. These three concepts are generated from the given operational input models, including Petri nets, stimulus models, reconfiguration automata. A step contains an `action` of type `PostGenerationAction`. During generation, all reconfiguration actions and term templates have been resolved beforehand.

After test execution, for each `TestSuite` a `TestReport` is created, which contains `TestRun`s relating each to one specific `TestCase`. Respectively, the result of executing a `TestStep` is a `TestStepRun`. Each of the latter has a `Verdict` with a given name (e.g., PASS or FAIL) and, optionally, a comment. Furthermore, a verdict stores the `state` after the step's execution in form of an array of instances of `Atom` to allow for tracing failures more exactly.

5.4. Test Generation Framework

Whereas the metamodel implements the syntax of the formal concepts, the components of the generator realize the semantics. An additional requirement is that engineers should be enabled to build in semantics of custom stimulus. Based on the stimulus model interface (cf. Section 4.3.6), the basic syntax can be extended by custom elements. In case of additional semantics, the interpretation mechanism must be designed in a similar, flexible approach.

The proposed solution to this challenge is illustrated in Figure 5.11. The top-level type is an instance of `Interpretor`, which is an iterator over `InterpretorState`s. The latter type maintains a queue of instances of `State`. Queued states are those, which have already been discovered and whose children have to be computed to further traverse the state space.

Each state knows its parent and assembles a set of `Atom`s (cf. Figure 5.4). In this way, each detail of a state is defined completely in the form of simple propositions on feature activation, value assignment, etc. Furthermore, a state can store an `entryAction` to indicate which communication has taken place when it was reached.

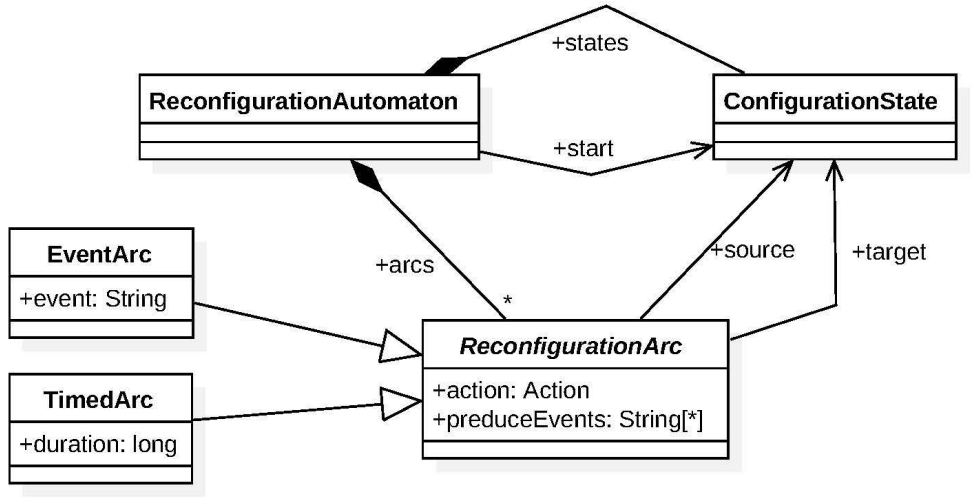


Figure 5.9.: Common metamodel for automaton-based stimulus models reconfiguration automata (package `reconfiguration`). *An automaton aggregates several states including a start state. Arcs are constraint by conditions and perform actions. In stimulus models, time-controlled arcs are used; in reconfiguration models event-controlled ones.*

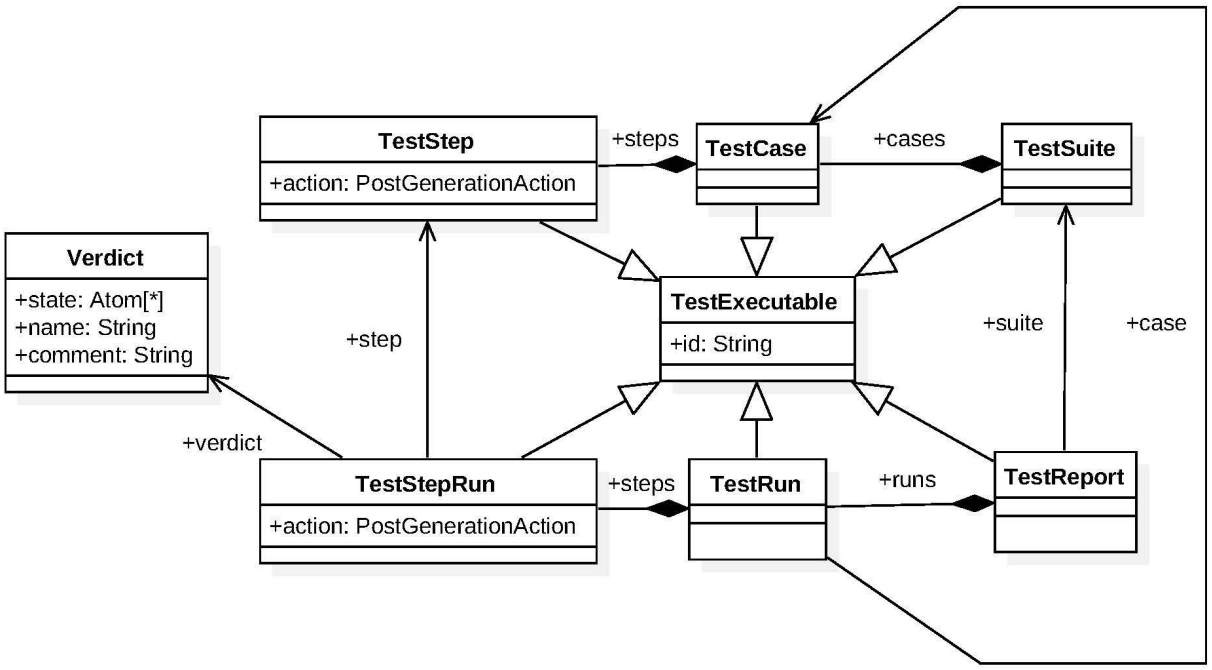


Figure 5.10.: Metamodel for test-suites (package `test`). *A test-suite is a set of test-cases, which are sequences of test steps defining a certain action. The results of test execution are test reports, which are sets of test runs, which are sequences of step runs. The latter additional contains a verdict including the reached state after execution. Each report element points to the executed test-suite element. All elements of test-suites and reports inherit from the test executable class, which allows a uniform handling in test replay.*

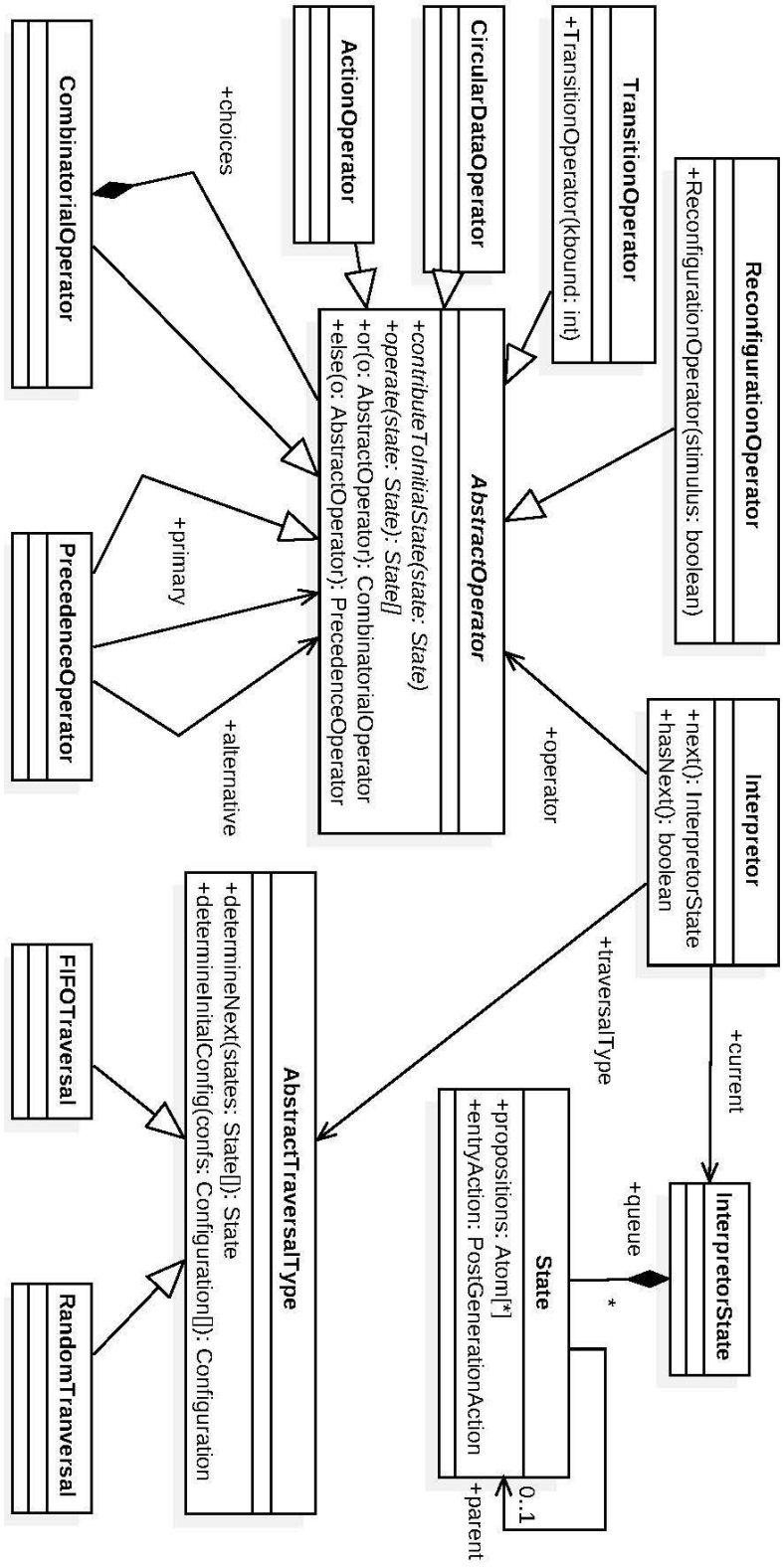


Figure 5.11.: Classes and relations of the generator framework (package `generator`). *The semantics of actions, Petri net transitions, and reconfiguration automata are each implemented in specific operators. Operators can be flexibly combined or ordered by combinatorial and precedence operators. An operator generates child states from a single parent state, whereas each state is composed of atoms and has an entry action. The interpreter iterates over interpreter states. A traversal type defines the exploration strategy through the state space.*

States are reached from each other by letting operators change certain propositions. For this purpose, different semantics are implemented in specific operators, whereas each operator inherits from class `AbstractOperator`. In the framework, operators for actions (`ActionOperator`), Petri net transitions (`TransitionOperator`), reconfiguration automata (`ReconfigurationOperator`) are included. Having an abstract operator and multiple specializations that can be exchanged spans a strategy pattern [GHJV94], which allows for flexible designing a custom generator for newly introduced model parts.

The latter two operators' constructors accept arguments for setting specific parameters in their semantics. For Petri nets, the modeler can set a k-boundedness constraint, which restricts the state space. Furthermore, the constructor of `ReconfigurationOperator` accepts a parameter that defines whether the stimulus model semantics or reconfiguration semantics should be used.

`CircularDataOperator` provides a second stimulus operator example for interpreting the wind and illumination model as introduced in Section 4.3.6. The designer of a custom stimulus model is free to provide additional operator implementations.

During interpretation, several operators reason on the basis of atoms that comprise the current state. For this purpose, operators can be built, which check satisfiability or constraint problems from the treated model part, which are compiled together with the given atoms and handed to Sat4J to be solved. For instance, `TransitionOperator` builds an SAT problem from adaptive Petri net's application conditions as well as an SAT problem from the net's structure and `TokenAtoms` to derive which transitions can be executed next. Furthermore, in the same way, a satisfiability problem is derived from the defined feature tree, which can be checked after each operation step. Thus, the user of the generator can cut the execution sequence as soon as invalid variants are reached.

Two methods are required in each operator implementation: (1) `contributeToInitialState(...)` gives a model-specific operator the chance to derive atoms for an initial state from the given models. For instance, from Petri nets `TokenAtoms` and `TimeAtoms` are derived. (2) `operate(...)` performs the actual transition from one state to a set of child states.

Besides model-specific operators, the framework user is additionally equipped with two operators that help to define the interpretation workflow. `CombinatorialOperator` composes multiple operators so that they operate on the identical state and their results are aggregated. Based on this procedure, for instance, several different operators for stimulus models can be treated with equal precedence. To define a different rule of precedence, the class `PrecedenceOperator` can be used. Its semantics is as follows: First, the `primary` operator is executed. If it returns a non-empty set of child states, the operation is done. Otherwise, the `alternative` operator is executed. The approach gives the possibility to chain operators. For instance, the framework user can decide to consume all produced events before producing new ones so that stimulus models are given a higher precedence. Basically, these two pseudo-operators structure a chain of responsibility and can be created via a *fluent interface* by the methods `AbstractOperator.or(...)` and `AbstractOperator.else(...)`. The following listing shows an example configuration (Java code):

```
1 public AbstractOperator createOperatorChain(){
2 return new ActionOperator().else(new ReconfigurationOperator(false))
3  .else(new ReconfiugrationOperator(true).or(new CircularDataOperator()))
4  .else(new TransitionOperator());
5 }
```

Listing 5.7: Using the fluent interface for chaining generator operators.

In this example, the configuration prescribes that actions have to be treated before all other syntactic elements. Next, reconfiguration automata are executed with the goal to consume events in the queue. Afterwards, both stimulus operators are combined. In this step, potentially new events are generated. Finally, the interpreter shall execute Petri net transitions.

For simulation and generation, the interpreter determines, which trajectories through the operator-generated state space are relevant. In simulation, only one trajectory is considered as the real system state is taken in the loop and its real state is reflected. In generation, different coverage strategies may be implemented. Here, the heterogeneity of the employed models makes it hard to define a single appropriate strategy. Thus, the MATE framework avoids to solve this question for arbitrary model types and only provides an interface and some default implementations. For this purpose, the class `AbstractTraversalType` prescribes two crucial operations. At initialization, there may be a set of feature configurations given, from which one should be selected. This task solves `determineInitialConfig(...)`. During state space exploration, method `determineNext(...)` is in charge to select the next state to be further expanded by the operator chain.

Potential traversal strategies that are specific to a concrete SASuT can be implemented based on this interface. As a generic strategy, a `FIFOTraveral` type is built in, which selects states always by their order in the interpretation queue. Alternatively, `RandomTraversal` selects a random state to continue with from the already explored ones. When the `Interpretor` is initialized, a user-selected traversal type has to be provided.

5.5. Test Automation Framework

MATE requires technology-specific connectors to adapt simulation and test-cases to a specific SASuT. To foster the technological independence of the test environment, a connector framework is provided that equips test engineers with hooks and interfaces for automating test execution.

The class diagram in Figure 5.12 depicts the automation framework's components. Technology-specific implementations of `AbstractionConnectorType` are aggregated by the singleton class `ConnectionManager`, which provides a method `register(...)` for adding new test-automation connectors. Each type further aggregates a set of `Connector`s, each representing a connection to an individual SASuT. The automation process can `reset(...)` or `terminate(...)` a connection. An action can be passed to the method `automate(...)`, where the connector translates it to technology-specific commands. Furthermore, an instance of `Responder` is required. For actions, where the simulation state is manipulated depending on sensed information (i.e., in instances of `InTheLoopAction`) this responder methods assign values to properties (method `setData(...)`) and to (de-)activate features (method `setFeatureState(...)`). Another option to retrieve information from the SASuT is the method `event(...)`, which allows for inducing events from the test-automation connector. These events can then be consumed by reconfiguration automata.

The framework provides two generic implementations of automation connectors in package `automation.basic`. Both are typed by the singleton class `BasicConnectorType`. The most general implementation is `ManualConnector`. When connected, it performs all actions by showing dialog windows where the tester confirms `TermAction`s or selects values for instances of `InTheLoopAction`. In the case of `GetAttributeValueAction`, only values of the domain of the attribute are accepted. For `GetFeatureStateAction`s, the user is asked to determine whether the respective feature is currently active or not. When connecting to a `RandomAutomation`, value input is created by selecting a random value from the domain and feature state is determined randomly, whereas all `TermAction`s are confirmed without interpreting them. Both implementations allow for testing a model without connecting to a real SASuT but instead to an automatically created mock. All connector implementations can inherit from `ManualAutomation` so that actions that cannot be interpreted are delegated to a manual (i.e., human) interaction.

The illustration also shows an implementation of a connector for the drone system. An extra `ConnectorType` implementation is introduced as well as a `Connector`, which adapts test actions to the drone's remote service interface. Although the depicted classes are based on Java, MATE enables testers to load connector implementations from Scala files, which have several advantages. Firstly, Scala has features of functional programming and is, thus, well-equipped for evaluating

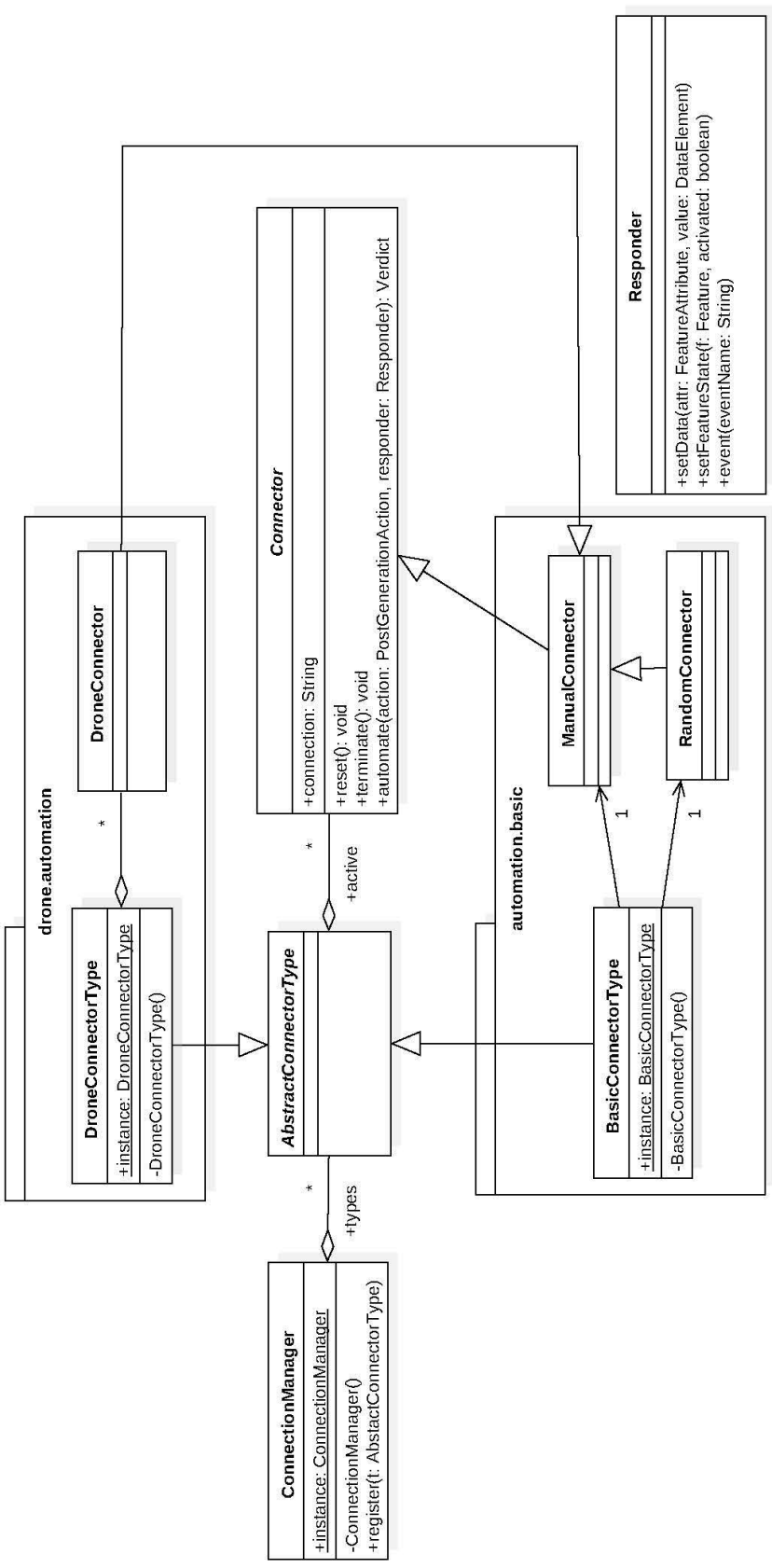


Figure 5.12.: Metamodel for test-automation (package `automation`). *A connection manager aggregates technology-specific automation types, which further aggregate connectors for types of SASuTs. The connector interprets actions and manipulates the simulation state via a responder. Basic implementations for manual and random interaction are built in. For the drone example, a new connector hooks into the framework.*

messages and trees of terms. Secondly, and even more important, it can be compiled when the test model is loaded into the interpreter and, thus, also be changed with the test model. This feature allows for co-evolving the test model and the test driver so that test-automation actions, which the tester misses at the time of modeling, can be implemented on the fly without the need for a new MATE plug-in.

The automation framework is, besides the generator, another component in the intermediate layer of MATE. On this level, also the model editors are provided. All editors in MATE are graphical or, in case of the variability model, tree-based. Together with the interactive simulator, editors enable the test designer to run a systematic test process. This process and the utilization of MATE within it are discussed in the next section.

5.6. MATE Tooling and the SAS Test Process

MATE provides tooling in a homogeneous technical space along the complete dynamic test process. According to standard ISO/IEC/IEEE 29119, dynamic testing incorporates four crucial sub-processes, which are requirements to the MATE toolchain:

> Standard processes of dynamic testing according to ISO/IEC/IEEE 29119:
>
> 1. **Test Design & Implementation Process:** A test specification is derived, which contains test-cases and adequacy criteria. Test cases to be executed for a specific objective are assembled to test procedures. Furthermore, test environment requirements are defined.
> 2. **Test Environment Set-up & Maintenance Process:** The test environment (i.e., test bed) is established and later maintained, which results in a test environment readiness report or, respectively, a test environment update.
> 3. **Test Execution Process:** The specified test procedures are run in the established environment. The actual results are recorded in this process and compared to the expected ones. The outcome of this procedure is reported in form a test execution log.
> 4. **Test Incident Reporting Process:** After analyzing the test results for failures, an incident report is generated as feedback to all responsible stakeholders.

Each component of MATE fulfills a certain step within this standard process. Because MATE performs MBT, step (1), test design, is leveraged to the modeling of context variability, stimulus, behavior, and adaptation. Also, test-case generation can be understood as part of test design, which is, however, automated at this point. To support step (2), test environment setup and maintenance, MATE provides its automation framework. Step (3), test execution, means to run tests and observe the outcome. Depending on the concrete scenario, in these tasks covered by executing generated test-cases or interactive simulation. Finally, for step (4), MATE generates reports that give an answer about the quantitative relation of passed or failed test and all recorded information including traceability information.

From the modeler's perspective, MATE's overall workflow specializes the iterative process from the previous chapter (cf. Section 4.3.2), which is illustrated in Figure 5.13. Depending on the concrete setup, the entry point to this process can vary. For instance, the test modeler may start with modeling the general behavior of the SASuT by a Petri net without considering any context. In another scenario, it could be beneficial to first build a test driver to gain knowledge about what actions can be tested run-timeively from the test model.

However, because the test process is designed in an iterative fashion, its steps can be reached independently from the concrete entry point. As an extension to the conceptional process, the modeling phase incorporates the specification of the proposed model types: adaptive Petri

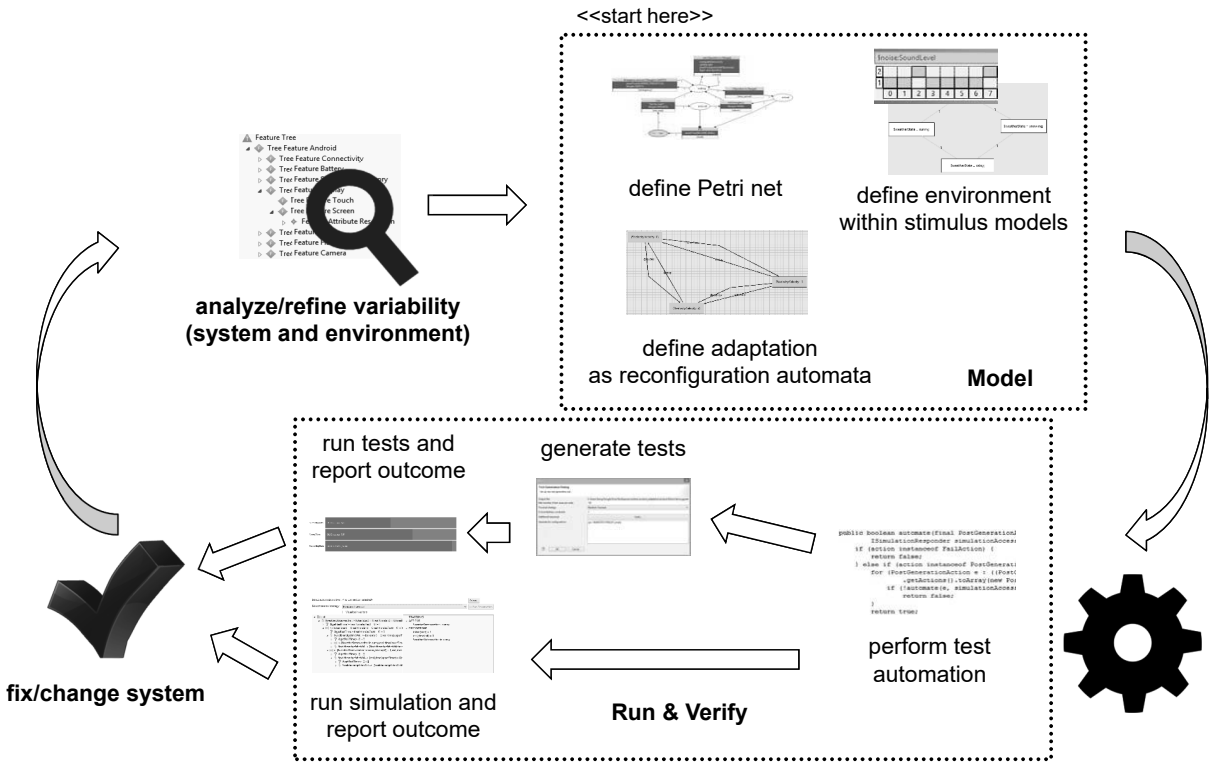


Figure 5.13.: The process of SAS testing with MATE. *Firstly, variability models are defined. Secondly, the behavioral models are set up: Petri net, stimulus, and reconfiguration. Thirdly, a test-automation adapter is built. Fourthly, tests are generated or simulation is performed. Based on the test or simulation outcome, the system can be fixed. For regression and refinement, modeling can be iterated as often as necessary.*

nets, stimulus models, and reconfiguration automata. During the the Run&Verify phase, test-automation, and simulation, respectively generation and test execution, is performed.

The presented process requires tool support, including a user interface for modeling and reporting, implementations of the metamodel as well as algorithms that implement the generation and simulation functionality. In the following, these components are discussed in detail.

5.6.1. Test Modeling

Initially, the test modeler is required to analyze the test object and its environment. All information assembled from this analysis has to be specified as model instances of the MATE metamodel. For this purpose, editors for tree-like structures and graphical editors are provided, depending on the targeted model type.

EMF generates tree-based editors automatically so that the modeler only has to put the minimal additional effort in this task. This especially holds for the feature-based variability model for context and adaptation modes. Furthermore, test-suite and report models are tree-structured, and so their creation and editing are performed within tree-editors.

Graphical editors fit well for graph-based models, such as reconfiguration automata and Petri nets. This editor type is based on GEF but customized for the specific notation. All textual elements of the graphically designed model instances are given to a jParsec implementation of the concrete syntax, which creates instances of the abstract syntax. The environment signalizes failed parsing to the user by coloring red the affected model elements. Thus, graphical and textual modeling seamlessly integrate within a common interface.

Besides adhering to a certain concrete syntax, some additional validations on the model can

be performed. For instance, is has to be checked whether all name properties are correctly set and all automaton-based models haven a start state. For this purpose, modelers define rules, which analyze the complete model instance after each editing step. The editors give feedback again by coloring the corrupted elements red.

For the purpose of creating custom stimulus models, the test engineer has to provide a notation and an editor as well. The generation of tree editors produces the least effort within EMF, whereas graphical editors must be programmed manually and laboriously.

5.6.2. Test Case Generation

After the successful validation of the designed model instances, the generator is ready to produce test-cases. For this purpose, an `Interpreter` instance is factorized and parameterized with the set of all models to be considered. The interpreter's operator chain may contain certain operators with model-specific settings to restrict the search-able state space (e.g., a k-boundedness constraint for the Petri net `TransitionOperator`). Additionally, a `TraversalStrategy` and a maximum number of test-cases is defined to direct and constrain the generation process.

In the beginning, the generator collects all predefined configurations and selects one of them by utilizing the traversal strategy. After producing all test-cases for this configuration, the generator restarts the process with next variant.

During generation, the interpreter iterates over the state space; the precedence of the states to be considered next is controlled by the given traversal strategy. As soon as one state is qualified final (i.e., no child state can be derived), the path to this state is back-traced along the `parent` attribute and entry actions are sequentially compiled to a test-case. The following listing shows the complete generation process as Java code:

```
1  public void generateTestCases(AbstractTraversalStrategy ts, int
       maxCases){
2   Interpreter i = new Interpreter();
3   i.setOperator(createOperatorChain());
4   i.setTraversalStrategy(ts);
5
6   while(i.hasNext()){
7    InterpretorState is = i.next();
8    State s = i.getTraversalStrategy().determineNext(i.getQueue());
9    State[] next = i.getOperator().operate(s);
10
11   if(next.length == 0){
12    produceTestCase(s); //backtrace parents and store steps
13    if(--maxCases == 0)return;
14   } else {
15    for(State n:next){
16     i.enqueue(n); //store derived states
17    }
18   }
19  }
20 }
```

Listing 5.8: Strategy-based algorithm for generating test-cases.

After setting up the `Interpreter` instance, the iteration loops starts and utilizes the constructed operator chain to produce children from each found state. If no `next` states are found, a test-case is generated. Otherwise, the all children are enqueued and considered later. The parameter `maxCases` restricts the generation process to the maximum number of test-cases to be produced.

The result is a tree-structure of test-suites, each associated with an initial configuration, test-cases, and test steps. The complete structure can now be executed by a test-automation con-

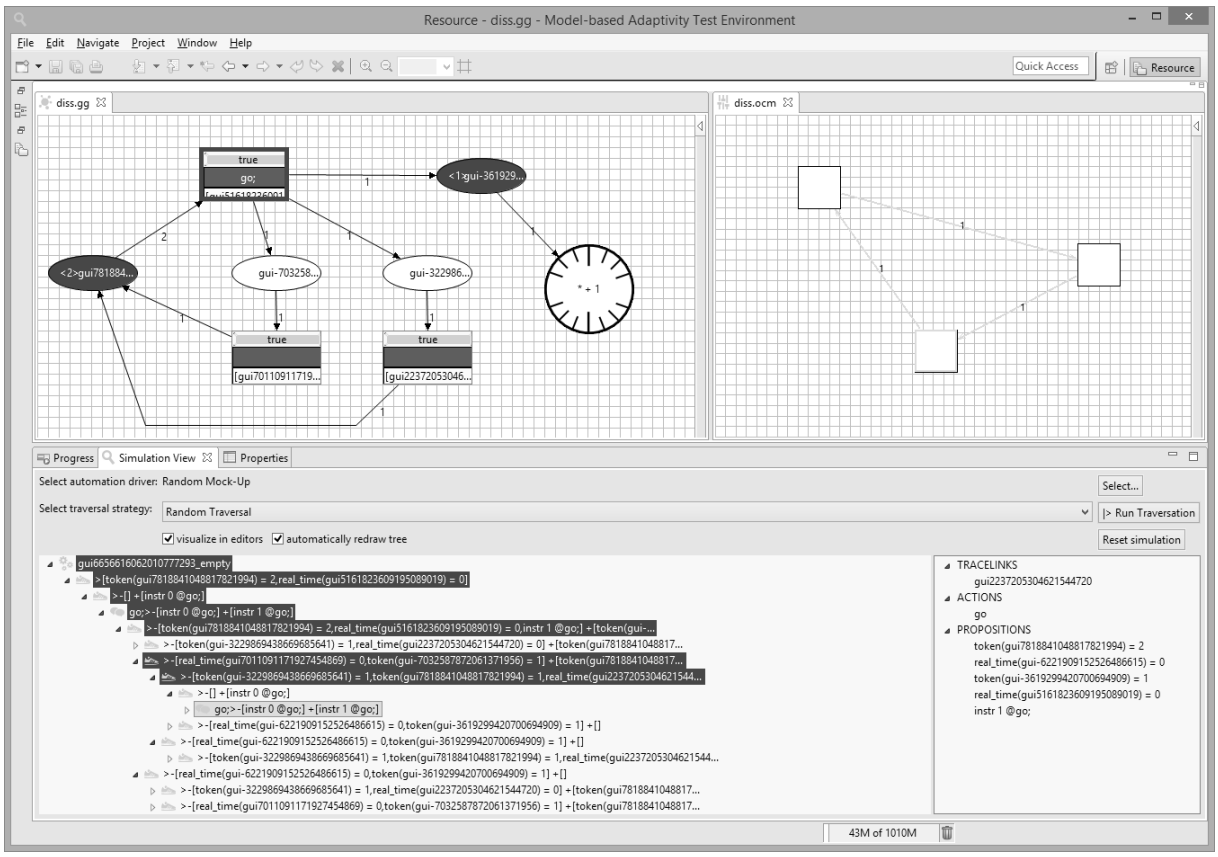


Figure 5.14.: A running simulation in MATEs. *In the simulation view, the user selects a test-automation connector and a traversal strategy. The explored state space is presented as a tree. The executed path within the tree is colored in blue as well as the model state in the graphical editors. Detailed information as trace links, the entry action, and set propositions are shown right in the outline. Simulation can be paused, restarted, and reset at any time.*

nector against the SASuT's interface.

5.6.3. Test Case Execution and Test Reporting

In test execution, steps are executed against the test-automation connector. For this purpose, each test step is simply passed to the `automate(...)` method of the provided connector. As `Responder`, a null object is passed because ITL testing is only possible in simulation.

After the action was performed, the test step is mapped to an equivalent `TestStepRun` by cloning the action for reporting. Furthermore, the executor adds a verdict, which contains a copy of the current state. Step runs constitute a test-case run, which further constitutes test reports.

Not only test-suites but also test reports can be executed to enable replaying tests. In this manner, the tester can periodically execute tests to uncover regression.

Finally, a statistical analysis is performed by measuring the relation between different verdict types like PASS and FAIL. To support this evaluation, bar charts are shown, which illustrate test success and, thus, indicate the degree of correctness of the system.

5.6.4. Interactive Simulation Frontend

In comparison to test generation, the simulator only executes one single trajectory through the state space. In this process, the last-executed state represents assumptions on the real state

of the SASuT at the same point in time, which allows for exchanging information between the simulator and the real test object and, thus, ITL testing.

The simulation process is *interactive* because the user may take direct manual control on it. Figure 5.14 shows MATE's user interface, which provides the necessary control for this interaction. The simulation view allows for parameterizing and directing the simulation. The user selects a test-automation connector and a traversal strategy. Setting these parameters enables to completely automate the simulation without further user interaction. Alternatively, the tester himself can take control of the traversal. For this purpose, he uses a tree-based representation of the state space. When clicking on one state node, the set-up operator is instrumented in the background, and the node's children are expanded. The outline on the right presents properties of a selected state in detail. This presentation includes trace links, the state's entry action, and the currently valid propositions (i.e., atoms). In parallel, these propositions are distributed to all opened editors, which may enrich the shown models to indicate the selected state. For instance, in the shown figure, the Petri net's places are colored in blue and labeled with the number of tokens within the given marking.

The user can execute one path within the state space tree. In this process, he double-clicks one node, which is immediately executed by the selected test-automation connector. The executed nodes are colored blue. Nodes that are not in the subtree of the last executed node cannot be run anymore from this moment, whereas they are still expandable. If manual execution is selected, the user will see several dialogs, which ask him to perform certain test actions and give feedback as well as the verdict to be returned.

If a traversal strategy is set up, the tree view determines, expands and executes automatically one path. The user may pause, re-start or reset the simulation at any point in time.

Interactive simulation as provided by MATE not only enables for ITL testing but also is a very helpful tool to avoid test-case explosion. For test modelers, the execution times that are caused by complex models can only be foreseen partially so that combinatorial execution times are often missed. The resulting problem is that the test-case generator produces a multitude of child states at single points which either slow down the generation process or prevent termination completely. To avoid this problem, the user can use the simulation view to step by step walk through the state space and find these problematic expansions. Afterwards, the modeler may decide to change the test model or re-design the traversal strategy accordingly.

5.7. Summary and Discussion

In this chapter, MATE has been described as a reference architecture that implements the proposed concepts and metamodels for SAS testing, including respective generation, simulation, and test-automation frameworks. Additionally, MATE provides a tool landscape, which equips the test engineer with editors and graphical control elements for performing a dynamic test process. The concrete modeling and verification process of MATE can be performed in an iterative manner, which leads to a growing level of detail concerning the considered variability.

The metamodel's implementation splits the introduced formalization into a set of graphical models, tree-structured models, and abstract syntax. Graphical models are especially adaptive Petri nets and reconfiguration automata. The latter metamodel additionally works as an automaton-based implementation of the introduced stimulus interface. Variability models for context and adaptation modes, as well as test-suites and reports, are tree-based. Abstract syntax models describe all concepts for textual notations, which include variability constraints, functions, terms, and test actions.

The operational semantics of the metamodel is implemented by operators, which hook into the generation framework. Operators span a chain of responsibility to organize the interpretation process. The interpreter itself iterators over the state space and traverses through it. Hereby, it is directed by a given traversal strategy.

The test-automation framework enables to run produced test actions against a real SASuT. Test engineers can instrument this framework and register custom connectors, which adapt to different technological spaces.

All those components support MATE with adhering to a complete standard dynamic test process. The task of specifying test-cases is replaced by modeling, which takes place in appropriate editors. The generation algorithm is then run, and test-cases are produced. These test-cases can be executed based on the previously programmed test-automation connector. Alternatively, simulation allows for interactively discovering the state space of the SASuT and ITL testing. MATE supports this process by visualizing the reachability tree, which can be explored by the tester step by step.

Both test-case generation and simulation are performed within a homogeneous infrastructure. Generator and simulator basically execute the identical algorithms. The only difference is that the simulator only executes one path instead of producing test-cases from all reachable path. This duality creates a high degree of reuse in all editing, interpretation, and test-automation components.

The complete structure of MATE framework fosters extensibility. Not only new test-automation connectors can be registered and, thus, new technical interfaces adopted, but also the engineer is equipped with an interface and a framework to build custom stimulus models. The restriction in the process is the need to adhere to the stimulus model interface. After defining a new metamodel and notation, new operator implementations can be introduced that interpret the newly added concepts. Even new adequacy strategies for these domain-specific models can be hooked-in based on implementations of the traversal strategy interface. As a default, a k-boundedness constraint is available in the Petri net transition operator for limiting state space exploration.

One drawback of the reference implementation is that parallelism is not supported. Test case generation on a multi-core system can run faster if the interpreter's iterator pattern is extended and certain trajectories are explored in parallel. However, MATE fulfills all discussed requirements and, thus, is quite powerful in comparison with the tools developed in related scientific work.

Part III.

Evaluation

6. Experimental Study: Self-Adaptive Co-Working Robots

A central aspect of SAS is their capability to decide autonomously, which is a valuable feature in service and industrial robotics. For the latter application area, autonomy is especially relevant in automated production sites, where part of the work is performed by humans and other parts by robots. However, nowadays factory automation processes are quite inflexible and conservative regarding context-awareness, safety, and the strict separation of human and robotic workplaces.

More flexibility in factory automation is a declared goal of the *Industry 4.0* paradigm [Sch16]. Besides mass-customization, robot co-working is an important key point. This vision of Industry 4.0 foresees that human workers directly cooperate with autonomous manipulators, for instance, by jointly processing wrought products, helping each other with heavyweights, measurement, or passing parts between work stations. Instead of nowadays robots working behind safety fences to avoid any human contact and injuries, new robotic devices offer the capabilities that allow for co-working with humans in the near future.

Lately, company and industrial research came up with first exemplars of sensitive robots. For instance, the KUKA LBR iiwa's motors include sensors that measure torque and, thus, can react to detected obstacles[1]. This industry-ready robot arm is based on the DLR Lightweight Robot, which is a result of the research of Haddadin et al. [HSF+11]. According to the authors, sensor-based reaction mechanisms are a fundamental ingredient of robots that are capable of directly co-working with human beings. Before this background, concepts, in which all robot decisions are controlled by a pre-designed environment model, are insufficient in co-working scenarios. Instead, the information that is gathered by sensors at run-time should be taken into account and analyzed by intelligent autonomous decision algorithms. In this way, the robotic application gets much safer, works with fewer errors, and is more flexible regarding adaptivity to changing contextual situations.

The DLR group also defines a set of global functional modes, which can deal as a basic concept of self-adaptation. The modes that the automation defines are switched based on the sensor input as Figure 6.1 illustrates. With humans absent, the robot operates in *autonomous mode*. As soon as the human is *in perception* of the robot's senors, one of the *interaction modes* is activated. Here, the authors distinguish between a *collaborative mode*, where the robot directly interacts with the human co-worker, and the *human-friendly mode*, in which further autonomous operation takes place in the presence of a human. In this mode, the efficiency of the application (e.g., its operation speed) must be decreased to lower the risk of severe injuries.

Furthermore, when a fault condition occurs, the *fault reaction mode* is activated, where the fault must be handled. This is especially the case if the perception of the human is lost and

[1] `http://www.kuka-lbr-iiwa.com/`

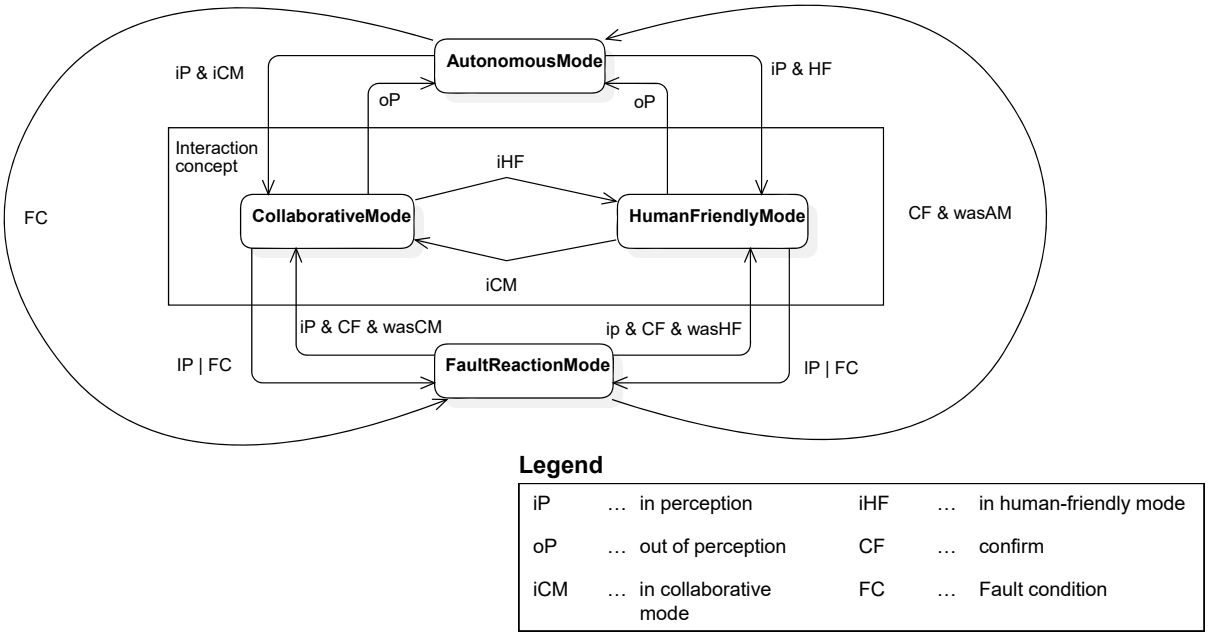


Figure 6.1.: Functional modes of the DLR co-worker, redrawn from [HSF+11]. *Four modes are distinguished, depending on the whether the human co-worker is in the perception of robot's sensors. If so, the robot switches from an autonomous mode to one of the interaction concepts, which let the robot either act collaboratively or with special (human-friendly) sensitiveness. Faults lead to a fault mode where safety is focused.*

the collision avoidance cannot work properly anymore. In the automaton, as it is depicted in [HSF+11], the fault mode can only be left with a confirmation.

Haddadin et al. discuss several solutions for each functional mode, such as trajectory scaling, path deformations, and soft-robotics based grasping as well as strategies for visual interaction. Besides those research questions, another challenge is to equip the robots with a set of sensors to recognize human presence and posture. For instance, optical tracking with stereo cameras or laser scanners are potential approaches. Alternatively, the project *WEarable Interaction with Robots* (WEIR) provides a solution, where body parts' positions are precisely monitored by intelligent clothing, which is equipped with inertial motion units (IMUs). The gathered orientation data of body parts is mapped to a human body model, which can be reasoned on to compute safe reactions to human movement. In comparison to optical tracking, WEIR has certain advantages: firstly, no visual overlapping occurs and, secondly, the system is completely mobile as it does not have to be installed in a fixed position within the factory building.

Originally, the WEIR system was built to record human movements and map them to a robot's kinematics. In this manner, automation processes should be prototyped quickly without any need for programming. The robot operator captures her/his movements and later replays them via a kinematic mapping on the robotic system. For a more complex use, the captured sequences can be assembled to an automated workflow, whose execution depends on the spatial location of physical system parts.

The workflow functionality of WEIR also provides a foundation for self-adaptivity. Force limits, speed, and autonomous behavior can be adapted to postures of humans and the robot itself. However, safety is critical in this kind of system, and the workflow designer should test it with caution. Consequently, in this chapter, a MATE-based test model for a WEIR-based case study shall demonstrate the applicability of the presented methods and realizations in a realistic scenario. The experiment shall incorporate a self-adaptive robotic system that uses the software and devices of the WEIR project to implement the behavior as proposed for the DLR co-worker. In comparison to the drone example from Chapter 4.3, the hereby elaborated setup is quite more

complex so that only excerpts of the complete test model can be shown. Despite this increased level of difficulty, the industrial robot requires being tested under the same requirements than the drone example, which provides strong support for the accuracy of MATE and its concepts.

The chapter starts with a more detailed introduction to WEIR in Section 6.1 and to its adaptation features. A WEIR-based application named Cinderella is discussed in Section 6.2. Afterwards, in Section 6.3, the required test models for this application are described. Finally, in Section 6.4, the study is put into relation to the problem set of this thesis.

6.1. Robot Teaching and Co-Working with WEIR

Controlling robots from manually written code is quite hard because movements have to be expressed in coordinates, motor control commands, or other hardware-related parameters. To avoid putting a disproportional effort into try-and-error programming, robots can be taught from human guidance.

Several methods have been approached for that purpose. One possibility is to guide the robot by direct manipulation. Recently, lightweight robots are equipped with torque sensors within their axes' motors so that they can measure externally applied force and navigate to coordinates where a human operator shifts the joints manually. However, this direct guidance approach works only for robotic applications that are accessible to direct human interaction. This approach neglects applications in hazardous, critical, or inconvenient environments like in chemical, nuclear, or semi-conductor industry. Furthermore, direct human to robot interaction is not well suited for heavy-weight robots due to the unacceptable risk of injuries.

Another method is teaching from human movements, which means that body postures are mapped to the robot's joint positions, motor angles, or translations. Using such a method allows operating robots from remote places, independently from their weight class and without any physical risk to the operator. Most approaches that implemented such human-body-based control make use of optical tracking, which is quite immobile and hard to set up. Cameras for body tracking must be installed in a fixed location or be calibrated very precisely.

To overcome the problems of direct manipulation and optical tracking, the chair of software technology at Technische Universität Dresden designed the WEIR system, which tracks human movements from wearable-mounted sensors and guides the robot from the gathered information. The WEIR system consists of a jacket and a glove, which contain IMUs and measure the orientation of multiple body parts. The captured human kinematics are mapped to robot kinematics by algorithmic components of WEIR software stack. Movements can be recorded and replayed so that teaching is only a matter of minutes. Thus, the WEIR system has a multitude of advantages in comparison to optical tracking, which includes minimal setup time, very low cost, and independence from environmental disturbances.

After recording, positions and movements are available as data sets, which can be composed to a workflow. A workflow consists of tasks, which each represent one recorded data set. Tasks are connected by arcs, which are conditioned by system states. WEIR's included workflow mechanism allows for quickly building prototypes of automation processes without programming.

Even more, workflows, the parameters of robot control, and the software infrastructure are adaptable due to the features of an adaptation plug-in. This component allows the definition of robot behavior dependent on the interaction with certain spatial regions with robot parts or human body parts. The plug-in's adaptation capabilities match well with the requirements tackled by MATE and, thus, are well suited to be tested by the use of the concepts proposed in this thesis.

Figure 6.2.: WEIR hardware. *Jacket and glove mount IMUs. Each part of the clothing contains a microcomputer, which collects data from the IMUs and connects via BLE. The central node within the jacket connects via WIFI to a server, where the WEIR software processes the data.*

6.1.1. WEIR Hardware Components

In the WEIR system, inputs are generated from a jacket and a glove, both equipped with sensors, microcomputers, and connectivity modules. Wearables and hardware components are depicted in Figure 6.2. The jacket hosts an Intel Edison module, which mediates the data flow between all nodes. Via a Bluetooth Low Energy (BLE) module, the Edison microcomputer connects to the glove and other potential clients. BLE offers a bidirectional data stream for interchanging measured sensor values as well as commands to the clients. Also, it consumes very few power so that the battery within each wearable component can supply the system for a whole working day.

The jacket also mounts three IMUs, which are all wired to the Edison board. One IMU is located close to the board itself to measure the orientation of the body's center in relation to an initially calibrated direction. Thus, a robot can not only work in the restricted operation space of the human arm but also next to and behind that workspace as soon as the human operator turns the direction of his torso.

Both the jacket's upper and lower arm contain each an additional IMU. The sensors are connected to the Edison node via solid cables within the jacket's lining. By fusing all three orientations (i.e., body center, upper arm, lower arm), and the fix lengths the body parts, the exact position of the wrist is computed.

The WEIR system uses the glove to determine the direction and state of a tool, which is attached to the actuator. In the experimental setup, this tool is a linear gripper, which can be used to pick objects and grasp them with a given force. The direction, where the tool is expected to point to, is derived from orientation data. For this purpose, the glove also mounts an IMU, which is cable-connected to a Flora Arduino board. This microcomputer was designed to be integrated into clothing and eases the use of conductive thread and the creation of washable electronics. Furthermore, a BLE module is connected to the board to allow for exchanging data with the Edison node. Besides the IMU, one finger (in other prototypes three fingers) holds a bendable resistor, a so-called FlexSensor. It measures the bending of the index finger, which is later mapped to the opening degree of the controlled gripper.

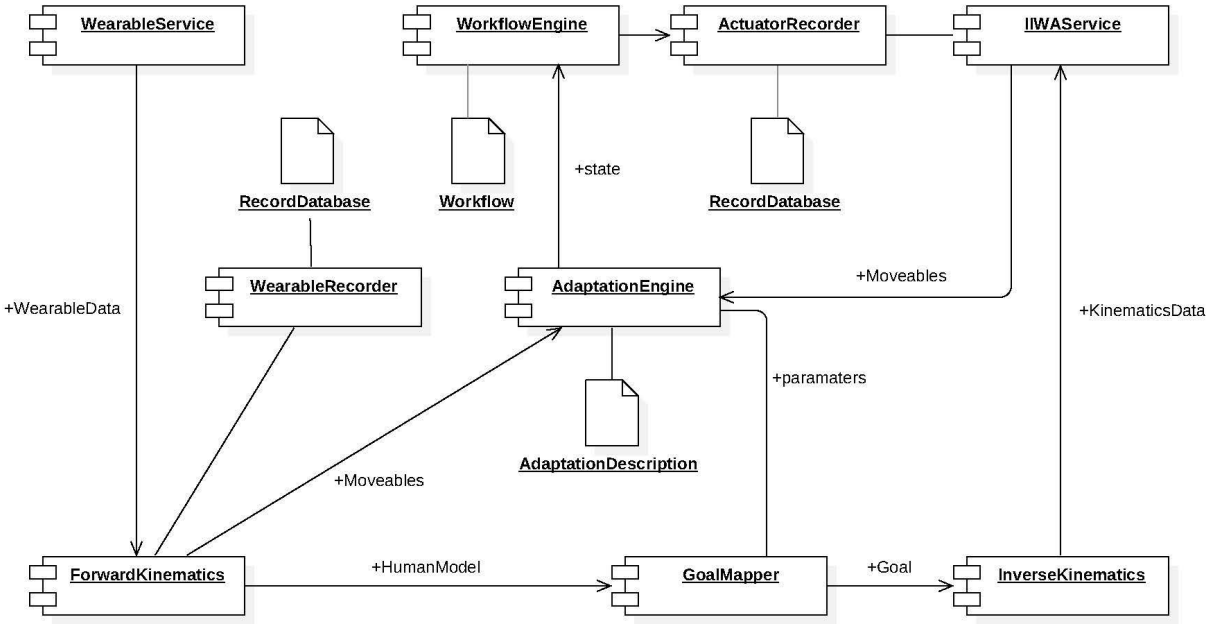


Figure 6.3.: WEIR software components. *The `WearableService` component receives data from the clothing, from which `ForwardKinematics` computes a human body model. The `GoalMapper` maps the model to a target point to be approached and `InverseKinematics` computes the robot arm's joint angles. Via the `IIWAService`, this information is forwarded to the actuator. The `AdaptationEngine` is responsible for adapting parameters and cooperates with the `WorkflowEngine`, which can run recorded steps autonomously. A workflow's steps are positions that were beforehand recorded by the `ActuatorRecorder`. Similarly, human movements can be recorded and replayed by the `WearableRecorder`.*

All IMUs deliver a measure of orientation so that a body model of the involved body parts can be computed. To correct this body model by an initially measured state, the human operator is expected to take a specific pose and calibrate before taking control of the robot. Afterwards, the gained model's location and orientation is absolute (i.e., all dimensions are aligned with those of the robot) and can be processed by the WEIR software.

6.1.2. WEIR Software Infrastructure

All mapping tasks besides other functionality can be decomposed in a set of software components as depicted in Figure 6.3. Because all the presented software components run on a central server, they have to connect to the wearables and as well to the robot. This task is performed by `WearableService` and `IIWAService`, whereas the latter is a connector named by the concrete actuator used in this setup.

The `WearableService` receives raw sensor data from the IMUs via WiFi. As a second step, the `ForwaredKinematics` component creates a body model from this data, which comprises three-dimensional vectors of the current posture of the human co-worker. From this vector model, the position and orientation of the hand palm can be derived and mapped to position and orientation (i.e., a goal) in the robot's coordinate system via the `GoalMapper` component. As a final mapping task, `InverseKinematics` computes the angles of the robot's axes that corresponds to the derived goal.

The data that is sent via the `IIWAService` contains angles for all joint axes and can be recorded by the `ActuatorRecorder` component to a `RecordDatabase` for later replay. A single robot posture or a sequence of postures (i.e., a motion) is the basis for tasks within a `Workflow`. The

`WorkflowEngine` executes the workflow step-by-step under consideration of adaptation-relevant events.

Besides robot postures, also those of the human can be stored and replayed later on. For this purpose, the `WearableRecorder` connects to the `ForwardKinematics` component. Instances of the human model, which have been generated during mapping, can be serialized within another `RecordDatabase` and used for analysis.

To permit self-adaptation, the `AdaptationEngine` alters several parameters depending on measured positions of so-called `Movables`. The latter are virtual representatives of physical objects that are located in a common coordinate system. In the experimental setup, the system is configured to inject position of all human body parts (computed by forward kinematics) and of the gripper. Because the WEIR data only delivers spatial coordinates in relation to the body itself, an API function allows inserting the assumed position of the body center within the robot's coordinate system. Thus, each body part can be located exactly in relation to the robot. For the gripper, `IIWAService` provides two movable instances: one is constructed by the measured state of the real gripper, and another one reflects the goal position where it is expected to navigate to. The `AdaptationDescription` defines three-dimensional bounds that change their state after a `Movable` enters or leaves the bounds. Based on this state, the workflow may react or the parameters of the `GoalMapper` are altered.

Besides the presented algorithmic components, WEIR additionally provides a dashboard for monitoring and configuration. The dashboard not only visualizes the recent spatial models of the robot and human worker, but also displays data rates of the communication channels between the server, the wearables, and the robot, including the gripper. Furthermore, certain actions, such as calibration and starting or stopping the recorder can be triggered from the dashboard user interface.

The architecture of the WEIR software can be classified component-based and event-driven because each component creates new event data that may be processed in isolation. Connections between components are established by an *Observer* pattern—each listening component registers at its data source [GHJV94]. The wearables produce events approximately every 100 milliseconds and, consequently, the robotic control acts in the same frequency. Due to its event- and component-based nature, the functionality of WEAR can easily be exchanged by replacing components or registering new observers.

6.1.3. KUKA LBR iiwa as WEIR Manipulator

In the presented experimental setup, the KUKA LBR iiwa 14 R820 is used as the actuator of choice, together with a linear gripper by Schunk. The robot arm is designed for tasks, in which objects up to 14kg have to be moved. Within this limit, the robot arm works with ca. $40mm/sec$ operation speed and $0.15mm$ repetition accuracy, which is well-suited for a broad range of industrial applications. The arm has seven joints, which each can rotate around their axes with different maximum and minimum angles. Depending on the concrete joint, the angles range from at least $-120 \ldots 120$ up to $-175 \ldots 175$ degrees. Because there is one more axis as required for full three-dimensional freedom of movement, the gaps within the joint's angle ranges are compensated—otherwise, the robot could not reach all points within its workspace. Using torque sensors, the LBR iiwa also supports sensitive reaction to externally applied forces. For instance, the robot arm can detect collisions with humans or objects or apply a limited force to a fragile workpiece.

The robot is installed on a rack (product name KUKA flexFellow), which contains a server, an air conditioning system, and energy supply. Different APIs allow monitoring and operating the robot. Provided programming languages are Java and C++, whereas the latter should be used if very little latency is required. The software system provides a real-time path planner that allows for performing inverse kinematics with guaranteed deadlines. The WEIR system does not make use of this capability to permit more parameterization of the algorithm. To communicate with

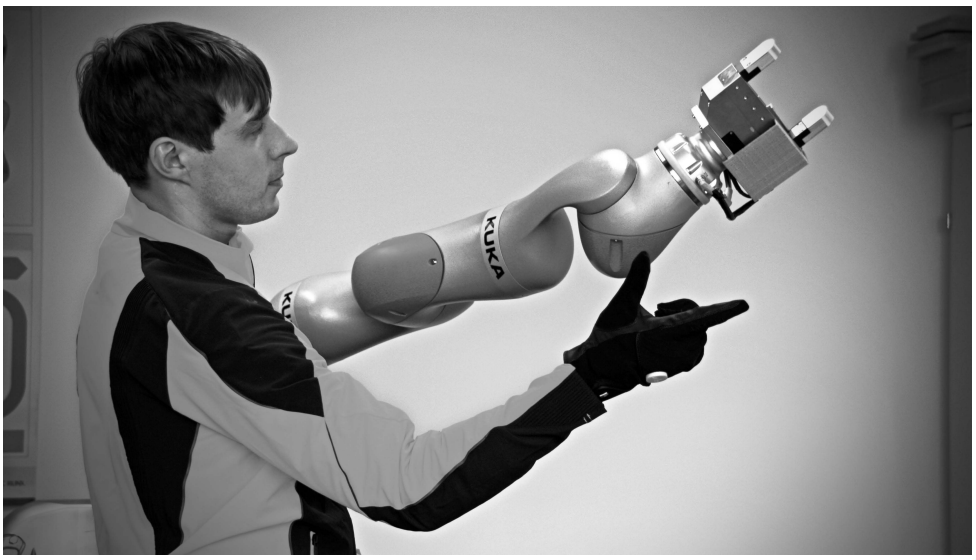


Figure 6.4.: WEIR-controlled KUKA LBR iiwa. *The hand palm's position and direction is mapped to the robot tool's coordinates, and the index finger controls the gripper's brackets.*

external systems, including WEIR, an Ethernet connection is available.

Another functionality, which the KUKA's software stack provides, is the capability to set up safety zones. Before starting any application, a set of immutable bounds is installed in the robot controller. As soon as an operation drives any part of the robot into those safety zones, the robot immediately stops all movements and activates its physical brakes. This safety feature prevents collisions with fixed objects within the robot's workspace due to wrong programming or manual control.

Because the robot has much more joints than the human worker, body parts are not individually mapped to the robot model. As depicted in Figure 6.4, the robot mimics the human hand's position and direction in relation to the body center. Furthermore, the brackets of the gripper tool are controlled by the bending value of the index finger. In this way, the human worker can navigate the robot to points within its workspace and let him grasp for objects. The software records these postures and movements to the database.

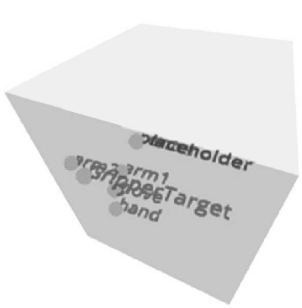
6.1.4. Self-Adaptation Capabilities of WEIR

The WEIR adaptation plug-in supports both parametric and behavioral adaptation. The monitored context results from two sensor inputs, which are (1) absolute positions of objects in relation to the robot's coordinate origin and (2) locations of certain movables (i.e., parts of the robot or human body). This spatial information is mapped to a common coordinate system, whose origin is a vertical axis through the robot's center. Whereas robot parts are already located in this coordinate system, human body parts are mapped there by using the functionality of the `GoalMapper` component (cf. Section 6.1.2).

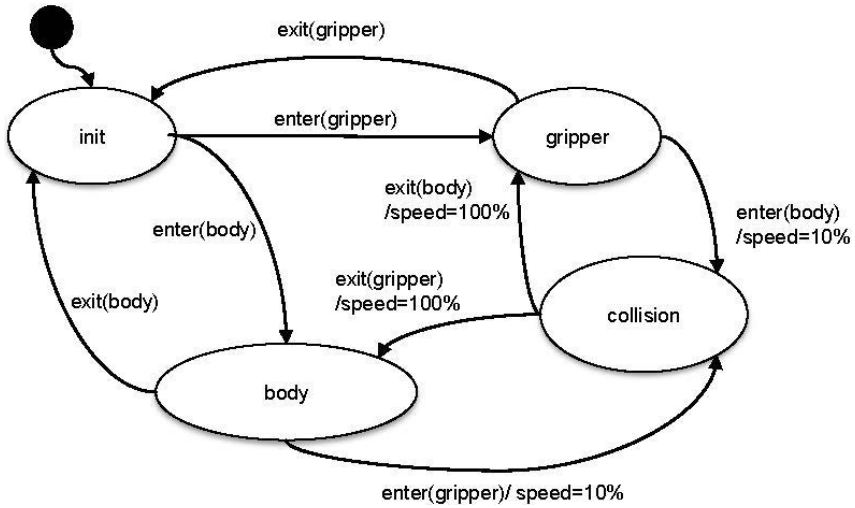
Based on this contextual information, the adaptation rules are defined within three-dimensional bounds. Those bounds allow for adapting parameters of the software and are also the basis for behavioral adaptation. Both adaptation levels are described in the following.

Adaptation Bounds and Parametric Adaptation

The introduced co-working mechanism derives adaptations for robotic control from spatial relations between human workers and the robot itself. In WEIR, such relations can be recognized by checking whether two or more movables locate in the same three-dimensional space or not.



(a) The WEIR dashboard component visualizes a box-shaped adaptation bound with movables that represent body parts and the targeted gripper location.



(b) `enter` and `exit` events of specific movables control a state machine, whose transitions trigger changes on parameters of the system.

Figure 6.5.: Adaptation bounds, movables, and state machine for collision detection.

Such spaces are defined in the form of *adaptation bounds*, which either are rectangular boxes or spheres. Figure 6.5a shows a dashboard screenshot, where a single adaptation bound is visualized. In the depicted case, only one box shape is installed, and the green points indicate current positions of movables, whose names indicate the locations of body part (arm1 and arm2 for upper and lower arm, center for body center, and hand for the glove). The spatial information for those movables is generated by the wearable devices. Furthermore, a `GripperTarget` is visible, which is the virtual position indicating where the gripper is expected to navigate to. Designers of adaptive behavior can upload files with spatial definitions to the WEIR server and, thus, install new spatial bounds for the design of adaptations at run-time.

Each adaptation bound relates to a state machine definition, whose transitions are triggered by either an `entry` event or an `exit` event of a specific movable. For instance, bounds can define a part of the robot's workspace, and the state machine allows for reflecting whether the robot operates in the same space as its human co-worker. For this purpose, the designer would declare a respective state machine with four states as depicted in Figure 6.5b: `init` (no movable entered), `gripper` (only gripper entered), `body` (only body entered), and `collision` (both entered). For all transitions that lead to the `collision` state, an action can be attached, which decreases the speed parameter of the robot driver; all outgoing transitions do the opposite.

A bound definition can also be understood as Event-Condition-Action (ECA) rule [BM09], where events are enter or exit signals, conditions are states of the bounds-related state machine, and actions are parametric adaptations. Additional to adaptation bounds and to reach to the complete expressiveness that MATE can test and simulate, a behavioral adaptation mechanism is required, which shall be described in the following.

Adaptive Workflows with Behavioral Adaptation

Whereas adaptation bounds provide a mechanism to define parametric adaptation, adaptive business logic can be defined in the form of workflows. The WEIR adaptation plug-in includes a workflow metamodel, in which transitions between workflow states are guarded by expressions over states of adaptation bounds.

The concrete appearance of a workflow is illustrated in Figure 6.6. Each workflow state has a name ($A \ldots D$) and plays a record from the record database. In the depicted example, the workflow starts with playing `fast.js` and repeats this operation as long as adaptation bound `boundA` remains in state `gripper`. Otherwise, the robot manipulator is retracted (state C, record

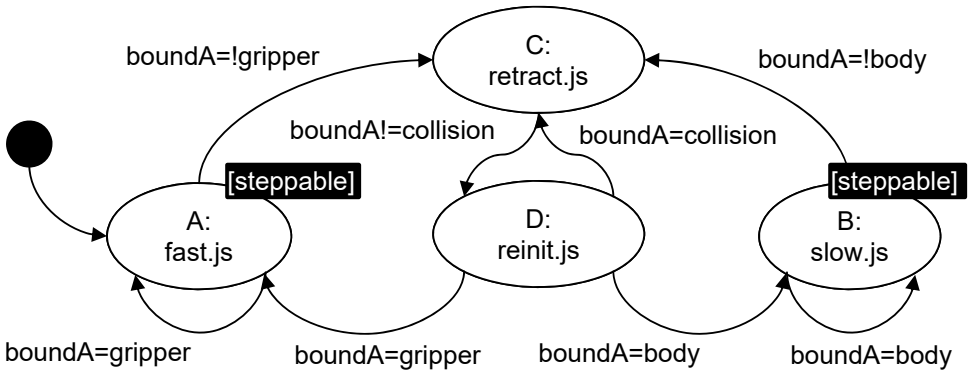


Figure 6.6.: Example WEIR adaptive workflow. *Workflow states play records from the record database and may be steppable for debugging. Transition monitor adaptation bounds and are activated if the annotated expressions evaluate to true.*

`retract.js`). If state `collision` is not active, the robot is re-initiated (state D) and advances with either fast operation for conditions with no humans involved or slow (state B). If a collision is detected in state D, the retraction is performed again.

The metamodel of workflows implements adaptive behavior. Expressions can also be formulated as conjunctions over multiple adaptation bounds to define more complex decision logic. While a workflow is run autonomously by the built-in workflow engine of WEIR, the adaptation bounds are monitored. After playing a record, the bounds' state is evaluated to determine the next task. In this way, both parametric and behavioral adaptation operate independently.

6.2. Cinderella as Testable Co-Working Application

Picking and placing objects with robots is an often demanded task in industrial automation and, thus, would benefit from being adopted in co-working approaches. The crucial challenge is to pick products from a factory line, transport them, and place them in new locations with high precision and in as little time as possible. In this section, a respective application is presented, which extends this task by self-adaptive robotic co-working. The task's workflow is taught and executed by using the WEIR system and its features targeting self-adaptation. In the first step, the robot is expected to pick objects from one of two source boxes. Second, the object's quality is classified and, depending on the verdict, the workflow chooses from one of two target boxes, where the objects will be initially placed, and put them into one of two other boxes depending on the quality of the respective object. Referring to the literary character Cinderella, who was in charge of performing this task manually, the application is named after her. In the following, the detailed setup of the Cinderella case study shall be discussed.

6.2.1. Cinderella Setup and Basic Functionality

Cinderella uses an experimental setup that allows for investigating its behavioral properties in an isolated test environment. Figure 6.7 depicts the robot and further equipment. The WEIR system is installed under the previously described conditions: the KUKA LBR iiwa combines with a two-bracket Schunk gripper, and the WEIR software runs externally on a notebook, which also hosts the dashboard. In front of the LBR's cabinet, a table is placed. On top of this table, within the robot arm's workspace, four boxes of equal size (ca. $30cm \times 30cm \times 20cm$) are put. The boxes represent the positions where the objects, which the robot would move in a production site, are located.

The two outer boxes (light gray) are the *sources*, from where the robot picks, whereas the inner boxes indicate two sink positions for sufficient (green) and insufficient (dark gray) quality.

The basic functionality is to pick from the sources with alternating order and place it in one of the sinks depending on an externally provided signal that indicates the quality.

In the presented test environment, several properties are simulated, including the quality verification signal, the exact grasping of physical objects, and the relative position of the human co-worker's body center. This functionality permits repeatability of the test-setup on the level of integration testing. Consequently, the test process can completely be automated and the co-working functionality is tested in isolation from other sources of failures, such as wrong grasping, insufficient object recognition, or erroneous factory logistics. All those features are neglected hereby to focus on testing self-adaptation in Cinderella.

6.2.2. Co-Working with Cinderella

Cinderella uses the features for self-adaptation of WEIR to behave accordingly to the human co-worker's position and movement. Figure 6.8a illustrates the self-adaptation and how it is set up. Without the presence of a human co-worker, the previously described sorting procedure is performed completely autonomously; otherwise, Cinderella adapts its speed and behavior to minimize the risk of potentially harmful physical contact. All functional modes of Cinderella directly map to those, which were defined by the DLR co-worker state machine (cf. Figure 6.1).

To implement the required behavior, several adaptation bounds are configured. The first self-adaptation aspect comprises a change of the robot's working speed as soon as the co-worker enters an `awareness` bound, which wraps the robot in a virtual safety zone. This adaptation bound is triggered when the human co-worker approaches. In this state, the robot moves with 30% of its maximum speed, whereas in all other cases adaptation the operation takes place with 90% speed. We name this behavioral requirement /**R1**/. The adaption of the robot's speed implements the transitions between autonomous mode to human-friendly mode (cf. Figure 6.1). Being inside the box triggers further parametric and behavioral adaptations. From a human co-worker's perspective, the robot reacts according to the following behavioral requirements, which are referred to in Figure 6.8a:

/**R2**/ If a human body part enters a box, in which the robot is currently operating, the robot movement is immediately stopped. This behavior implements a fault mode, which must be resolved before the operation continues.

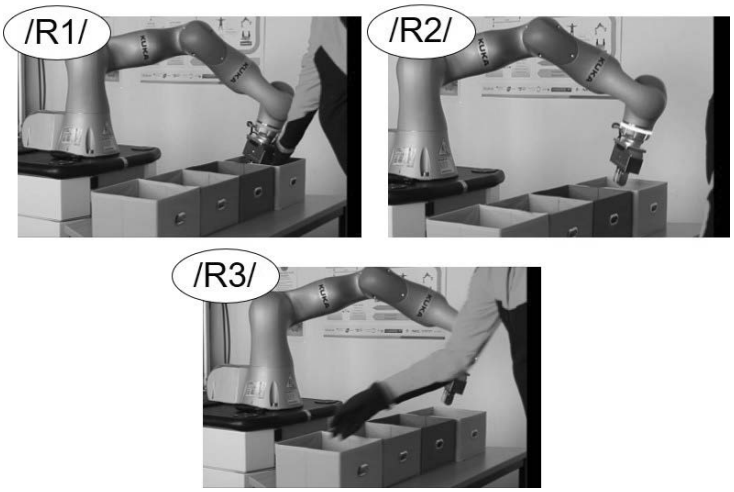
/**R3**/ At the moment, when the body part leaves the box, the operation is continued as planned. The fault mode is left and the human-friendly or collaborative mode is entered.

/**R4**/ If the human co-worker enters one of the source boxes, the robot chooses another box for the operation, so that both the human and the robot can work in a collaborative mode under reduced risk of collision.

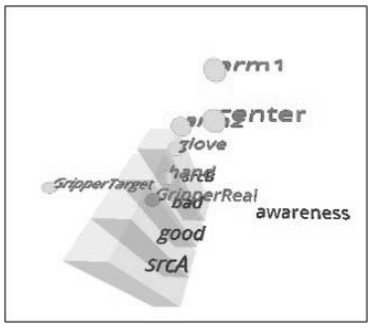
To configure the WEIR system for the described behavior, several adaptation bounds are defined. Figure 6.8b shows a dashboard screenshot, which visualizes those adaptation bounds. One virtual box (`srcA`, `good`, `bad`, `srcB`) each wraps a physical one. For safety reasons–because computation and pausation need time—, a virtual box is larger than its physical counterpart. Consequently, their maximum point in the z dimension (i.e., the height) is $65mm$ above the physical one and also all other dimensions are slightly larger. In this way, the robot is given some more time for stopping. All these single boxes are wrapped inside the `awareness` box, which marks the area, where the human-friendly mode is activated.

Adaptation bounds are configured with JSON; the Cinderella-specific definition is listed in Appendix 1. The root object is an array of bounds. Each box-shaped bound specifies its spatial location and size as `minPoint` and `maxPoint` properties. The state machine consists of an array of `transitions` with source (`from`) and target (`to`) states, whereas a state is represented as integer value. The initial state is stored in the `currentState` property. Furthermore, a transition refers

Figure 6.7.: Cinderella setup. *The robotic arm picks objects from the outer boxes and puts them in into one of the inner ones, depending on the object's quality: green for good, dark grey for non-acceptable.*



(a) /R1/ robot stops operation on collision, /R2/ continuation after the collision is resolved, /R3/ robot avoids human co-worker by using the alternative source box.



(b) Each source or sink box is encapsulated in a respective adaptation bound definition; an outer (hidden) *awareness* box is in charge to determine whether the co-worker is in vicinity.

Figure 6.8.: Cinderella co-working behavior.

to a movable, which triggers the state change. The property `isEnter` determines whether the enter or exit event of the referred movable is expected.

Similarly, the adaptive workflow is specified in JSON, which is listed in Appendix 2. In difference to state machines of adaptation bounds, the workflow consists of `tasks`, which perform actions. Each task defines a record (`posture`) to be played and whether it is `steppable`. For internal mappings, a task is referred to by its position within the task array so that the initial task `current` is 0. Besides tasks, there are `edges`, which connect a source (`from`) and a target (`to`) task. A `condition` is a logic expression. Each proposition refers to the examined adaptation bound (`boundsRef`) and the `state`s, which is expected. This proposition can also be a `negation`. Optionally, a `conjunction` may be added to specify a more complex expression.

Bounds configuration and workflow can be loaded at run-time so that it is even possible to exchange or alter running processes. A user can pause a workflow and resume it at any time. These capabilities make it possible to test Cinderella in a reproducible manner.

6.3. Testing Cinderella with MATE

Both parametric and behavioral adaptation, as Cinderella performs them, are testable properties. However, the stateful self-adaptation, which depends on contextual properties, creates a highly complex state space. The number of reachable states results from the combination of the current workflow state, the position of the human co-worker, her/his body posture, and the timely order of the occurrence of these signals.

MATE allows for modeling each of these aspects of Cinderella's self-adaptation. For this purpose, we adhere to the test process as defined in Section 5.6. The concrete entry point of the Cinderella setup into the test process is test-automation step. Afterwards, the variability model and the test model are created.

6.3.1. Automating Test Execution

Cinderella executes the installed workflow autonomously, which allows for completely automating test execution. As described in Section 5.5, MATE provides a framework for creating test-automation connectors. Both classes `AbstractConnectorType` and `Connector` have to be extended for this purpose in the form of a Scala-based implementation, which interprets the modeled actions and applies them in the SUT. The connector communicates via a *WebSocket*, which triggers actions within the running Cinderella instance. Available actions are the following:

- `set $human = ?`: Set the position of the human co-worker within the robot's coordinate system.
- `play(file)`: Play a previously recorded body movement from the `RecordDatabase` of the `WearableRecorder`.
- `calibrate`: Calibrate the recent body posture as the neutral (upright standing, hands on hip) viewing direction along the robots forward axis.
- `installBounds(file)`: Install adaptation bounds from a provided file.
- `startWorkflow(file)`: Start the workflow, which is defined in the provided file.
- `stopWorkflow`: Stop the current workflow.
- `assertSpeedMax(speed)`: Verify whether a maximum speed is not violated.
- `assertSpeedMin(speed)`: Verify whether a minimum speed is not violated.
- `assertPause`: Verify that the robot stopped completely.

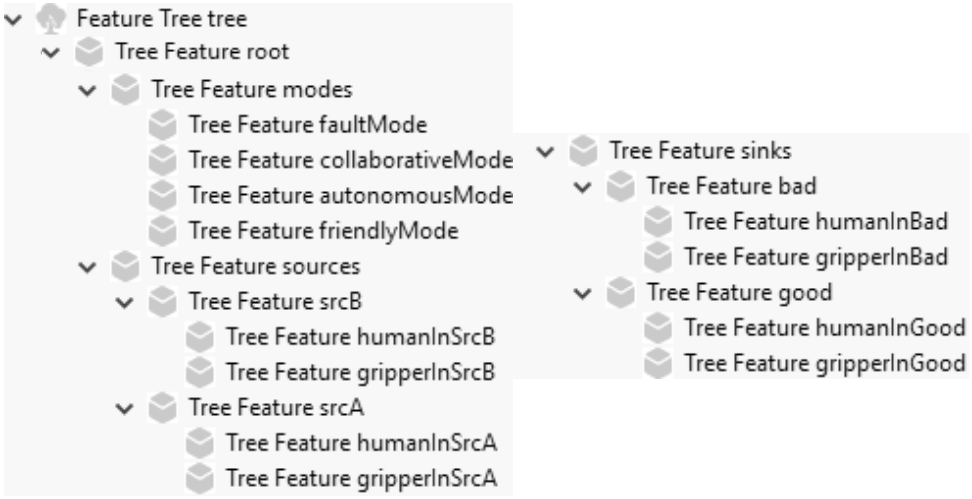


Figure 6.9.: Variability model for Cinderella. *The feature-modeled variability space defines potential properties of situations. It defines adaptation modes and situations when a certain movable entered a specific bound.*

- `assertEnter(boundsExpression)`: Verify that the provided adaptation bounds will be entered next by the robot gripper. Sends a step to WEIR and blocks until the verification passes or fails. A bounds expression is either a certain bound name or a disjunction of expressions denotes as `o(boundsExpression1, ..., boundsExpressionN)`, where `o` stands for inclusive or.

Using these actions, MATE is capable of simulating the co-working process and interactions with Cinderella. Speed, stopping, and order of actions (i.e., when a specific bound is entered) were chosen as the most important properties, which are relevant to verify whether safe co-working is ensured by the system. In the following, the Cinderella test models are discussed as well as how they make use of the connector's actions.

The driver also provides special functionality for ITL testing. For this purpose, events are produced by the driver as soon as the gripper enters a certain adaptation bound. Such an event is fed via the `Responder` (cf. Section 5.5) to the model at run-time. This functionality makes the test control flow dependent on the actual physical conditions and can only be used with MATE's ITL simulation, but not for generating sequential test-cases.

6.3.2. Modeling Cinderella in MATE

Despite Cinderella lets designers define self-adaptive behavior in the form of models, testers should re-model expected behavior of an SASuT from a black-box perspective. This allows for abstracting, defining scenarios, and searching for failures. The respective test models are derived from the requirements, which were defined in Section 6.2.2. In the following, all MATE-based test models for Cinderella are described.

Variability Model

In a first step, the variability in Cinderella is modeled within a context variability model. Whereas the application does not require static variability (e.g., product configuration before run-time), the tester should use feature trees to distinguish situations and adaptation modes. The result is the variability model as depicted in Figure 6.9. The `root` feature contains a `modes` feature, whose children represent all global adaptation modes of the co-worker automaton. Further children of `root` are `sources` and `sinks`, which contain for each type of box an individual subtree. Leaf features `humanIn*` and `gripperIn*` classify situations of human to robot interaction.

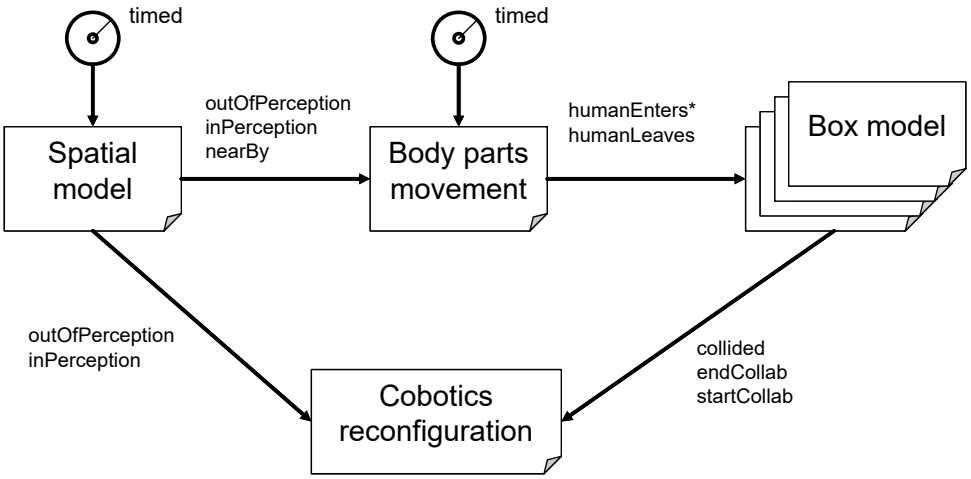


Figure 6.10.: Event flow through the Cinderella test model. *Spatial and body movement models are controlled by budgets of virtual time; whereas box models and the robotic co-worker's reconfiguration is controlled by events.*

The dynamic selection of variants resulting from this variability model is managed by context stimulus models and reconfiguration automata, which are presented in the following.

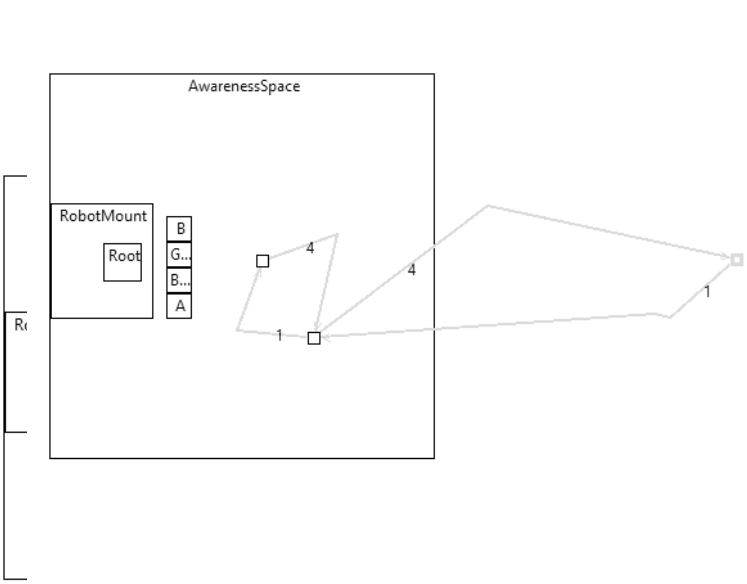
Stimulus Models: Absolute Spatial Movements

Context stimulus models are used to generate actions for manipulating the physical environment and the selected context variant virtually. In MATE, context stimulus models are controlled by budgets of virtual time and communicate with each other, or respectively, with reconfiguration automata, by producing or accepting events. As illustrated in Figure 6.10, four types of stimulus and reconfiguration models are created for Cinderella: firstly, a spatial model describes the movement scenarios of the whole body through the robot's coordinate system, and, secondly, the body part movement model defines scenarios of changes to the arm, hand, and the torso's rotation. These model instances define the context stimulus and are primarily controlled by time budgets. However, the body part model is partially controlled by events that are generated from the spatial model. Thirdly, each of the four Cinderella boxes is represented by a model of individual adaptation bounds. Fourthly, the robotic co-worker's reconfiguration is described as an automaton, which accepts events from the spatial model and the box models.

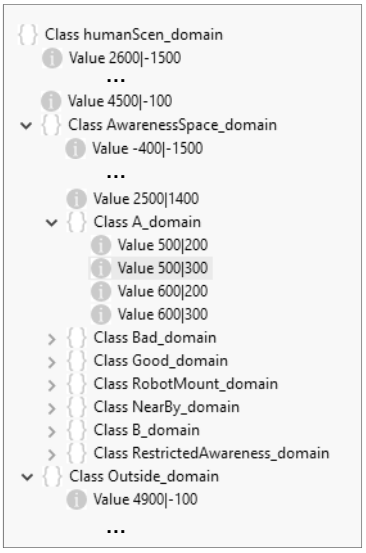
To decompose Cinderella's context parameters and enforce behavioral change, several events are predefined. The spatial model generates signals (`outOfPerception`, `inPerception`, and `nearBy`) that indicate where the simulated human co-worker is located. These signal events are consumed by the robotic co-worker's reconfiguration model to distinguish between autonomous mode, fault mode, and interaction modes. Furthermore, these events also cause changes in body parts movement model as certain human poses do no longer play a role when the awareness zone is left.

Additional events are produced during the change of poses in the body parts movements model. The events `humanEnterSrcA`, `humanEnterSrcB`, `humanEnterSrcGood`, and `humanEnterBad` (in Figure 6.10 summarized by a wildcard sign) indicate whether a certain adaptation bound has been entered and `humanLeaves` whether one has been left. The box models' output are events of type `collided`, `endCollab`, and `startCollab`, which trigger the co-worker automaton to either change in fault mode or between the two interaction modes.

Both the spatial and the body-part movement model adhere to a common interface as described in Section 4.3.6. However, they fundamentally differ in their graphical representation. The spatial representation has been developed to appropriately support modeling two-dimensional movement



(a



(b) Equivalence class tree generated from the spatial model.

Figure 6.11.: Spatial context stimulus model for Cinderella.

profiles. Figure 6.11a depicts the respective spatial model for Cinderella. Each rectangle defines a zone within the robot's coordinate system. Concrete zones are `AwarenessSpace`, which represents the adaptation to decreased speed (according to /**R1**/), `RobotMount` and `Root`, which set up the location of the robot and its mount, and one zone for each box. All these zones directly reflect adaptation bounds that are installed in Cinderella. Other, smaller zones are anonymous and represent areas, from which points in movement scenarios are generated.

Edges that connect some zones define potential movements with each a budget of virtual time. The most right zone represents the initial location (indicated by the green frame), whereas a location variable is set to when to model is loaded. For instance, with a budget of 1, this zone can be left and another zone inside the `AwarenessSpace` should be reached. From which location either the location right in front of the robot can be reached with a time budget of 1 or the simulated user is sent back to the origin with a budget of 4.

The model is parameterized by values for the scale and the shift in relation to its origin. Thus, the experiment's physical layout can directly be described by the spatial model and simulated during verification. Additionally, the modeler can define a stepping value, which is used to generate an equivalence class tree from the spatial model. The produced classes reflect the hierarchical structure of the two-dimensional zones and the leafs represent tuples of coordinates that can be set as the value of a location attribute. An excerpt of the tree, which is generated from the given model, is depicted in Figure 6.11b. The complete considered space is reflected by an equivalence class `humanScen_domain` and is the root of the tree. Each rectangle is mapped to a child class (e.g., `AwarenessSpace_domain`) in an equivalence class tree. Building such a structure from the spatial model requires all two-dimensional zones of one layer to be completely free of intersections. The leafs of the tree are coordinates that can be assigned to a variable `$human`. The coordinates are representatives of the equivalence class tree and have a minimum distance according to the set stepping value.

Entering a zone not only sets the `$human` variable to a representative value, but also may cause an event. For this purpose, the modeler can assign one or multiple event types to a spatial zone within the editor. Whereas the zone out of `AwarenessSpace` is assigned to the `outOfPerception` event, all zones inside `AwarenessSpace` are assigned to the event `inPerception`. An exception is the anonymous node right in front of the boxes, where a `nearBy` event should be signaled. These

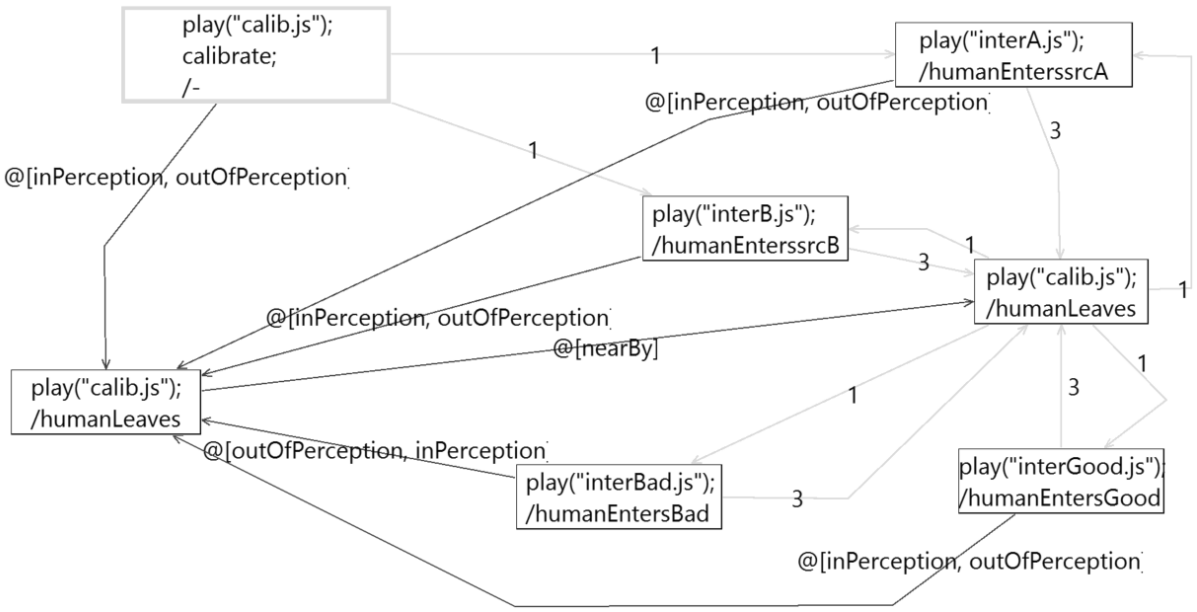


Figure 6.12.: Model of movement of body parts for Cinderella. *The initial green-framed node calibrates Cinderella. Nodes play records and produce events. Some transitions are controlled by virtual time, others by events from the spatial model.*

events are consumed by other behavioral models to define dependent adaptation or situational scenarios.

Based on the structure of the spatial model, the tester can design movement profiles. In the editor, paths are drawn between multiple spatial zones, which also may have branches. The generator is equipped with an operator that performs a depth-first search along those designed paths. From each spatial zone that is a node of the path, a representative is selected as coordinate. A found path of such coordinates results in a movement profile. Because the path elements can be parameterized with a budget of virtual time, the movement can be controlled by virtual time from timer transitions within an adaptive Petri net. Furthermore, it is possible to check whether the simulated user is located inside a certain zone by using the `InClass` function (cf. Section 5.3.2) within application conditions.

Stimulus Models: Movement of Body Parts

In contrast to spatial movements, the movements of body parts are much more complex and, thus, have to be abstracted in the Cinderella test model. For this purpose, the `play(file)` action is used. Body part movements are represented in a finite state machine, which mixes concepts of stimulus models and reconfiguration automata because it plays the role of an intermediate. Figure 6.12 illustrates this approach. The green-framed node is initially activated and its actions load and run `calib.js`. This record sets up positions of all body parts for calibration and triggers the calibration. Starting from the resulting state, two nodes can be reached, where the simulated body either interacts with source A (action `play("interA.js")`) or source B (action `play("interB.js")`). Afterwards, a respective event `humanEnterssrc*` is signaled. The posture is left by playing the `"calib.js"` record and signaling `humanLeaves`. From this state, also the sinks can be reached (`play("interBad.js")` and `play("interGood.js")`). Each of the named records was captured beforehand to deal as test data. The necessary time budget for entering a box is always 1, whereas for leaving a box a budget of 3 is required. Consequently, the simulator will always generate scenarios that contain more simulation steps with interaction than without. From each of the described nodes, an event-triggered edge ends at the calibration node, which can be understood as adaptation pause. The latter is always activated if the spatial model produces

an `inPerception` or `outOfPerception` event. In these cases, the simulated body is not placed right in front of the robot and does not interact. As soon as the interaction position is taken, the spatial movement model produces a `nearBy` event, and new body movements are generated.

Events synchronize spatial and body part movement models, so that body motions are produced within the simulated scenarios only in relevant situations. Events are broadcast to the complete set of models, which also includes the reconfiguration automata.

Reconfiguration Automata: Cinderella Boxes

For each Cinderella source or sink box, a reconfiguration automaton is provided. Figure 6.13 shows the respective instance for source box `A`. The initial bold-framed node deactivates the complete feature subtree (cf. Figure 6.9), which removes any information on the movables in the box from the configuration.

As soon as the gripper movable, respectively any human movable, enters the box—which is indicated by one of the events `humanEnterssrcA` or `gripperEnterssrcA`—the automaton switches its state. In case of the human movable, a new event `startCollab` is produced to signal that requirement /**R4**/ is now taken into account. In this condition, the robot is expected to collaborate with its human counterpart by using the alternative source box. If both types of movables enter the identical box, a `collided` event signals a dangerous condition, which must be reacted by stopping the robot's motion.

States can be left as soon as any event indicates that the human co-worker completely leaves the scene (`humanLeaves`) or the movable enters any other box (i.e., in case of the source `A` all events that end by `-entersBad`, `-entersGood`, or `-enterssrcB`). If the initial state is reached, the event `endCollab` is produced to signal the end of the collaboration.

Within this automaton, states incorporate entry actions for reconfiguration, more precisely, they activate or deactivate features, which indicate situations of movables. Whereas these situations are examined by the adaptive Petri net, produced events are consumed by another reconfiguration automation that reflects the adaptation mode of the robotic co-worker.

Reconfiguration Automaton: Cobotics

One more automaton is defined to directly reflect the robotic co-workers' adaptation modes as proposed in Figure 6.1 for the DLR co-worker. Figure 6.14 illustrates that the conceptional automation is directly mapped to states of MATE-based reconfiguration automaton. Each state reconfigures the feature-represented adaptation mode by first deactivating the complete mode sub-tree and, secondly, activating the respective mode feature.

The autonomous mode is considered to be the initial state of adaptation. When a human worker enters the awareness zone (event `inPerception`), one of the interaction modes is activated. Vice versa, if the human leaves the awareness zone (event `outOfPerception`), the automaton switches back to autonomous mode. More elaborate interaction is performed in collaborative mode if the `startCollab` event is signaled. As soon as event `endCollab` is detected, the co-worker falls back to human-friendly mode. Finally, each occurrence of `collided` leads to the activation of the fault mode, which is expected to halt the robot. This state can only be left by letting the simulated human leave the position in front of the robot, indicated by either the event `inPerception` or `outOfPerception`.

The coworker's reconfiguration automaton is the final sink for all events and determines the adaptation mode. This global state is used in the adaptive Petri net to decide which requirements should be asserted in a specific situation.

Adaptive Petri Net

Cinderella's application logic is reflected by the adaptive Petri net, which is depicted in Figure 6.15. Transition names are denoted at the bottom of each transition box. Initially, place

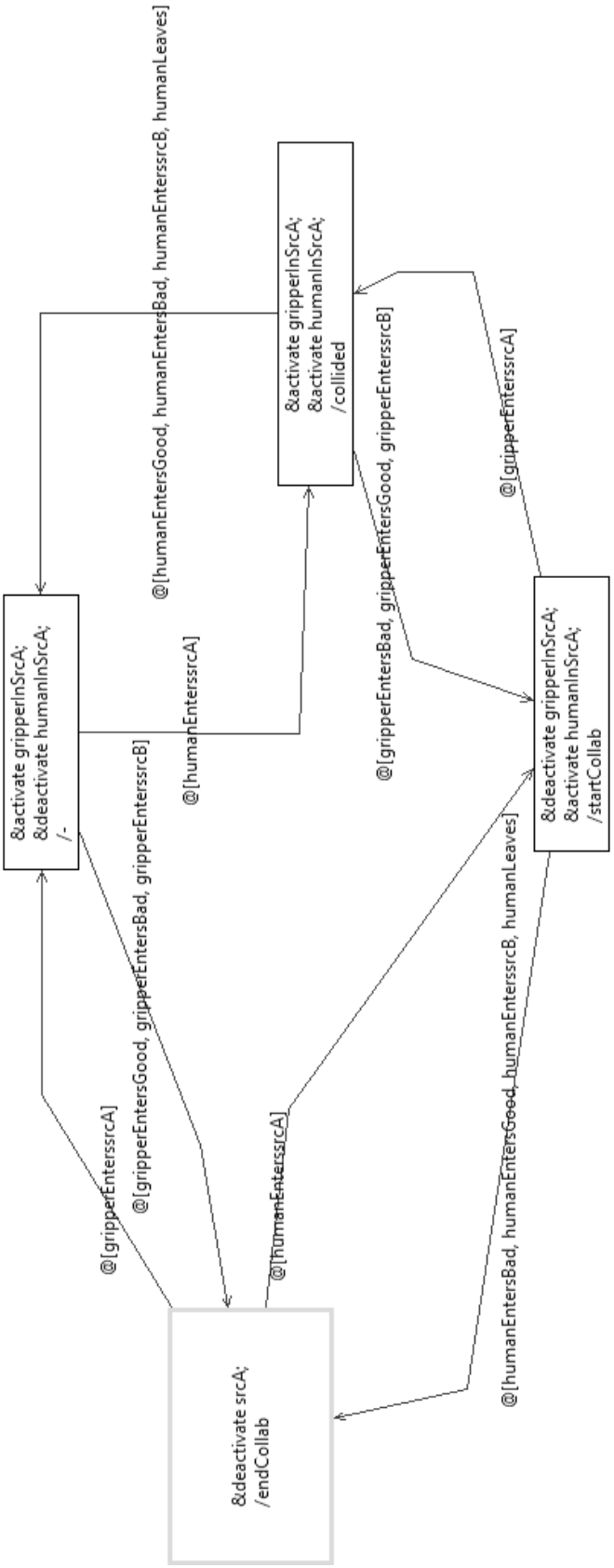


Figure 6.13.: Reconfiguration automaton for one Cinderella picking box. *Initially, no movables are inside the box. Events from body part movement trigger transitions. Entering a node causes feature selections that indicate the recent situation. Newly produced events are sent to the robotic co-worker's automaton.*

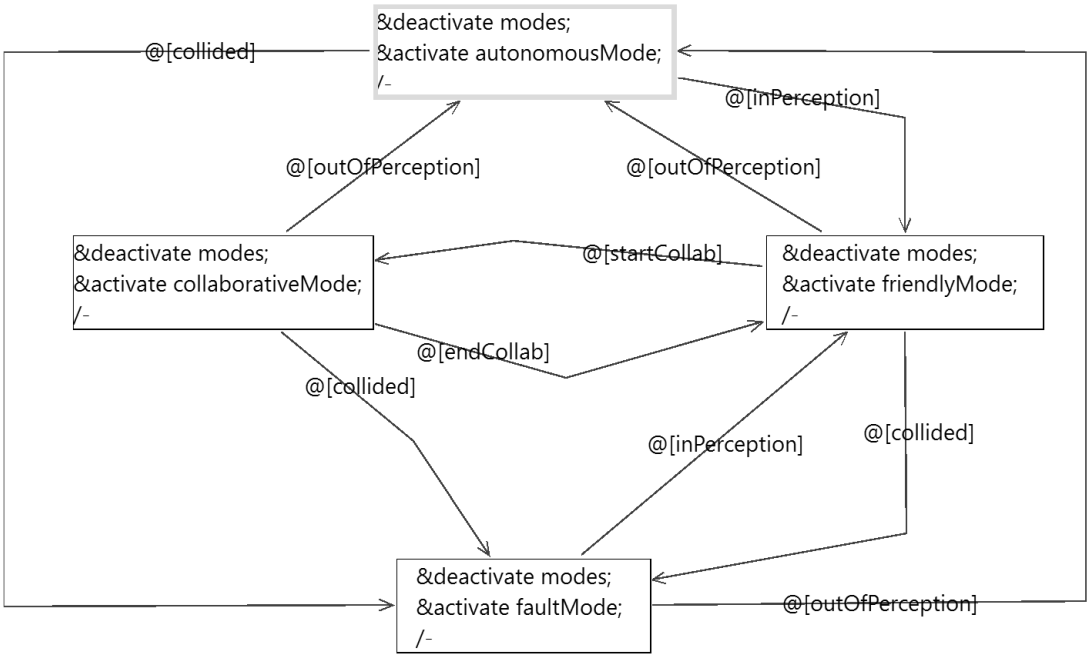


Figure 6.14.: Cobotics reconfiguration automaton for Cinderella. *Event-based edges switch between reconfiguration nodes. Each node activates a respective feature to indicate to current adaptation mode.*

`init` contains one token, so that transition `initialization` is activated. Additionally, there is one token in `A`, which reflects the assumption that the autonomous process always starts picking from `source A`.

Besides the presence of a token in `init`, the transition `initializing` is unconditioned because its application condition is always true. The first action `stopWorkflow` guarantees that no workflow is running and no interaction with previous test runs occurs. Afterwards, the correct adaptation bounds are installed from file `bounds-cinderella.js` as well as the autonomous Cinderella workflow from file `test.js` (cf. Appendix 7.2). Finally, a token is placed in `ready`.

The Cinderella application runs completely in parallel to the test model. Synchronization is reached by waiting for a re-entry of a certain situation (e.g., when the robot picks from `source A`), which triggers a stepping action in the test-automation connector (cf. Section 6.3.1). In that way, all actions of type `assertEnter(...)` will be activated and wait before an expected physical action is performed. Thus, the test driver runs in debugging-like mode, where each physical operation is first confirmed before another (autonomous) one is started.

From the place `ready`, the token is consumed by a timer transition, which adds a budget of virtual time of size 1 to all stimulus models (indicated by the wildcard sign `*`). Consequently, the spatial and the movement model for body parts are triggered. The spatial model switches to a position within the awareness zone and produces an `inPerception` event, which further activates the human-friendly mode (cf. Figures 6.11 and 6.14). Although the body-part movement model would now consume the time budget as well and play either actions that make the simulated human interact with one of the sources, the `inPerception` event lets it switch to a neutral position again (cf. Figure 6.12). Thus, in this situation, the body part movement model has no direct effect on the context situation. In the second round of execution, the spatial model reaches the `nearBy` position so that the movement model contributes to the context.

The next state establishes after storing a token in place `pick`. The activated transition depends on both the current adaptation mode and the currently preferred source box. The latter is determined by putting a token in either place `srcA`, which is the initial condition, or place `srcB`. To model this distinction, transitions consume from one of these places and put one token in the

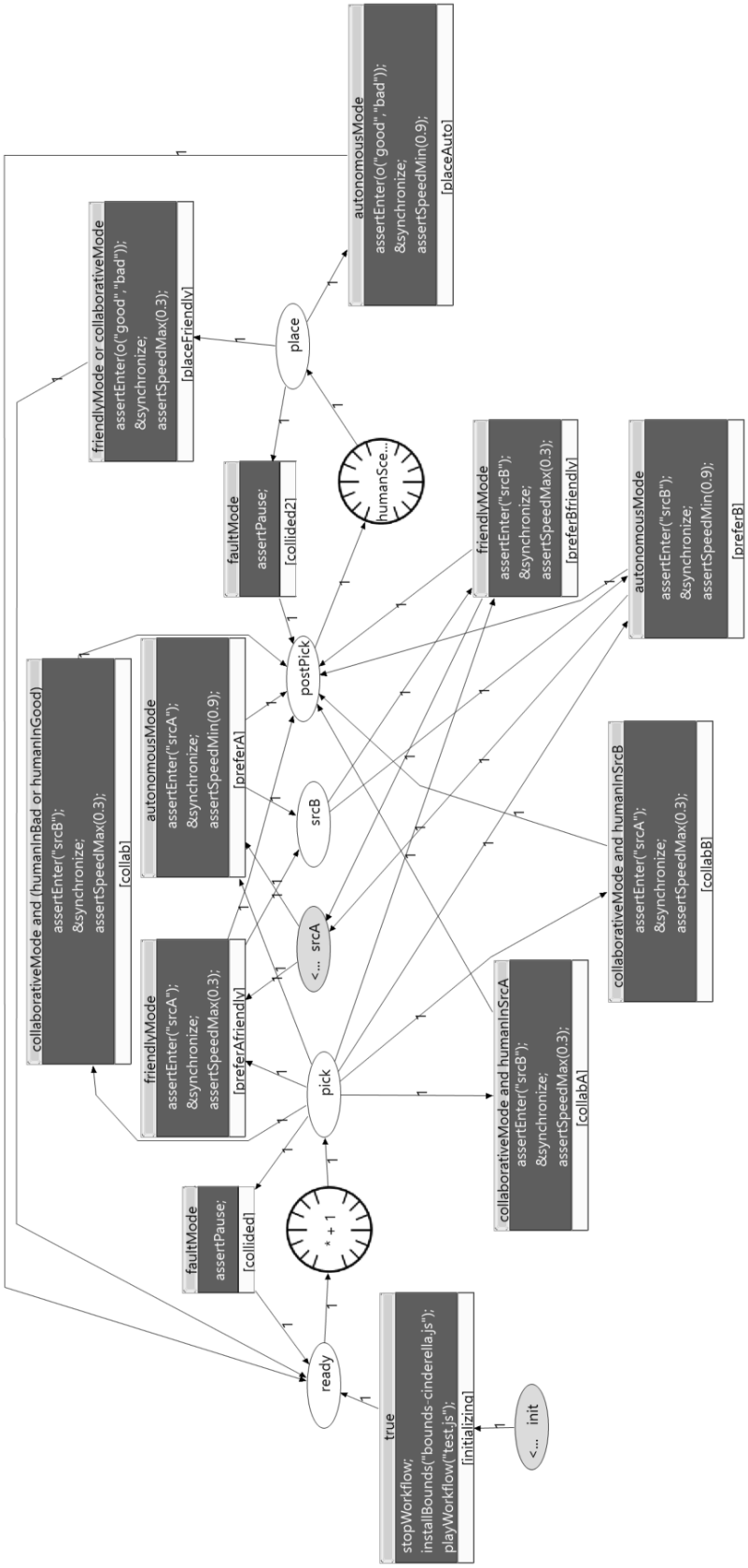


Figure 6.15.: Adaptive Petri net for Cinderella. *The application logic is modeled depending on the current adaptation mode. Thus, the active feature selection determines what is asserted. Two timer transitions control the stimulus models.*

other box place, which creates the alternating behavior. Exceptions are the fault mode and the collaborative mode, in which the source preference is ignored. Whereas in fault mode, no picking happens at all, in the collaborative mode the source for the robotic co-worker is determined by the current position of the human. This behavior is modeled by the application conditions of both transitions `collabA` and `collabB`. Furthermore, the fault mode is reflected by the transition `collided`. It asserts that the robot movement is paused and sets back the net to a state before executing the timer transition.

All other transitions that consume from the place `pick` perform assertions to verify the parametric adaptation in Cinderella. Firstly, the `assertEnter(...)` action is executed, which passes as soon as the connector detects that the adaptation bound belonging to the respective box is entered. If another adaptation bound is entered, the verification fails, and the test is canceled. Secondly, the events of type `gripperEnters*`, which are produced while waiting for the expected gripper state, are applied to the model. For this purpose, the action `synchronize` is executed. Thirdly, by running one of the actions `assertSpeedMax(...)` or `assertSpeedMin(...)`, respective functions within the test-automation connector are triggered, where the received messages are checked for the tolerated speed according to requirement `/R1/`. Finally, a token is produced and put into place `postPick`.

The following timer transitions steer the body part movement model so that we can also simulate another adaptation mode while placing the work piece into a sink. Outgoing from the place `place`, either the transition `collided` sets back the state for fault mode or the parametric adaptation is verified. Because there is no behavioral difference between the two interaction modes in the phase of placing an object to a sink, the respective branches of control mode can be checked equally in the transition `placeFriendly`. For autonomous mode, the transition `placeAuto` performs the verification. Both transitions do not distinguish between the boxes `good` and `bad` so that an or (i.e., `o(...)`) expression is provided as parameter of `assertEnter(...)`. Finally, the state is set back to `ready` to restart the control flow.

The adaptive Petri net models a counterpart of the actual autonomous workflow (i.e., the black-box behavior) and is synchronized to it. Control-flow synchronization is performed by waiting for certain positions of the robot, followed by transferring recognized events into the model at run-time. Both parametric (i.e., speed settings) and behavioral adaptation can be verified by determining concrete assertions depending on the robotic co-worker's adaptation mode. The complete set of test models is executed within an interpreter and its actions are applied by the provided test-automation connector, which is elaborated in the next section.

6.3.3. Testing Cinderella in the Loop

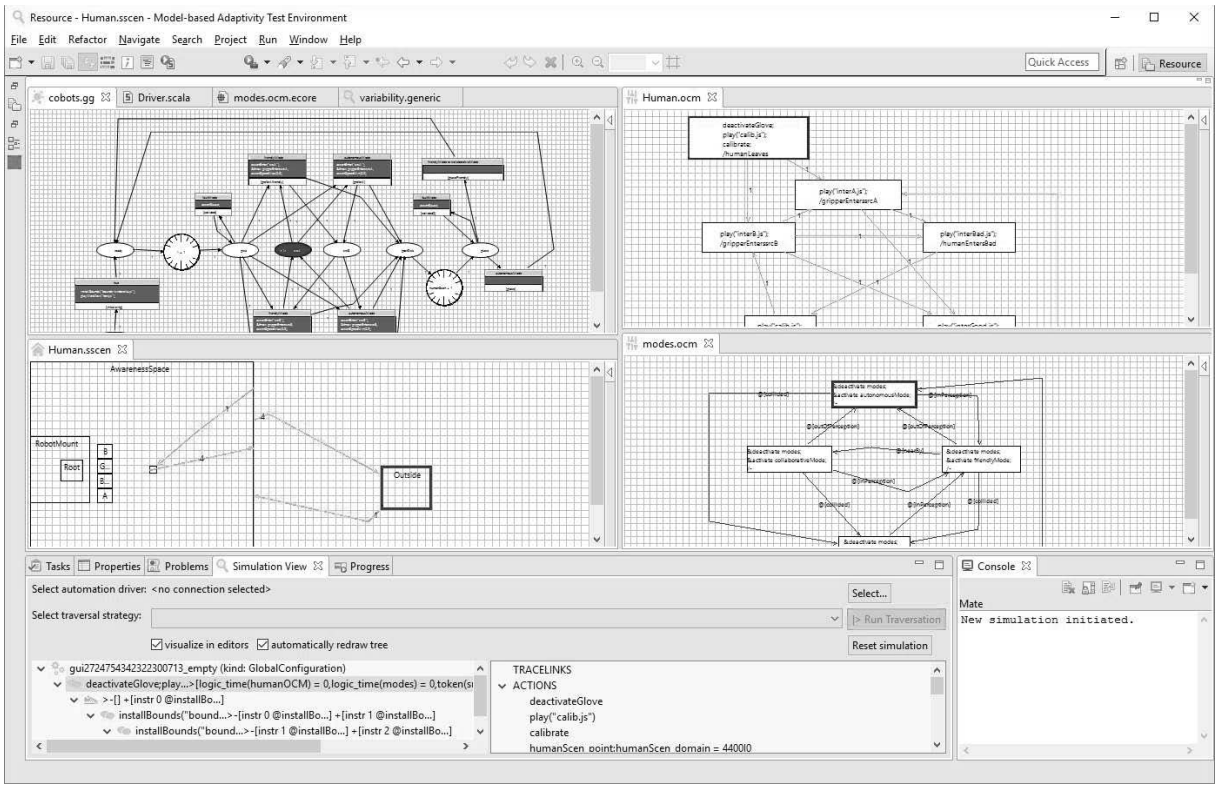
MATE interprets the introduced models in the loop so that the test can directly observe the corresponding virtual and physical states. An impression of test model execution is given in Figure 6.16. In Figure 6.16a, the robot is shown with the WEIR software running on a notebook. WEIR's dashboard visualizes the software' state and connectivity metrics. MATE runs in the background and communicates with the software. A beamer projects MATE's user interface state to the wall in the background.

In Figure 6.16b, the user interface is depicted in more detail. On the bottom, the reachability tree is unfolded. Each of the explored states can be mapped to the parts of the test model. For this purpose, MATE colors the active elements in blue, respectively with a bold frame. For instance, all places with tokens in the adaptive Petri net are marked as well as the current area within the spatial model, the active states of stimulus, and the reconfiguration automata.

Before starting the ITL simulation, the tester sets up the connector, which was programmed in Scala to co-evolve it with the actual test model. Furthermore, to automatically traverse the state space, a traversal strategy has to be provided. For this purpose, the default randomized strategy can be used in Cinderella.

With these settings, MATE runs the test process without any human intervention and only

(a) The robot operates autonomously while MATE interprets the test model and verifies movements.



(b) Zoomed-in MATE user interface. At the bottom, the reachability tree is explored according to a given strategy. Blue-marked model elements with a bold frame indicate the current state of the execution.

Figure 6.16.: Running Cinderella test.

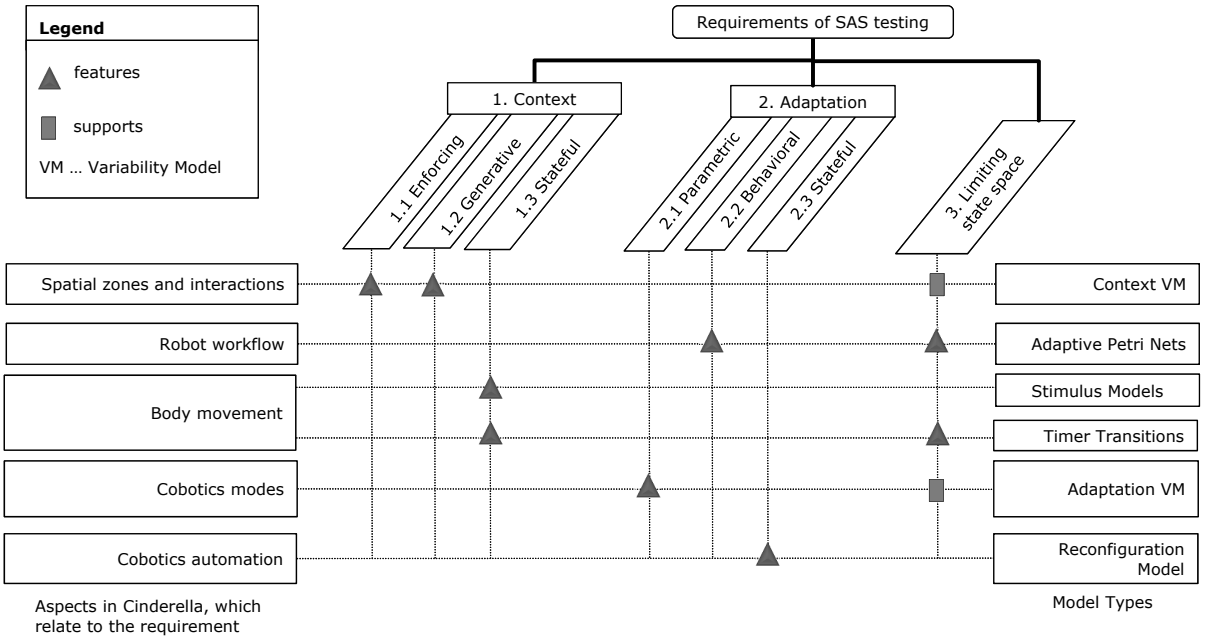


Figure 6.17.: Representation of proposed concepts in Cinderella.

stops if an error occurs. In this way, a long-running test can be performed, and the application's state space discovered deeply so that the designers reach a high confidence in their code's quality and safety.

6.4. Evaluation Verdict and Summary

The Cinderella application deals for an experimental study to demonstrate the effectiveness of MATE and its conceptional foundation. Consequently, the models, which were designed for testing the correctness of self-adaptive logic behind Cinderella, cover all proposed concepts from Chapter 4.3, as Figure 6.17 suggests. The illustration corresponds to the problem-solution fit, which was earlier discussed in Section 4.1, whereas conceptional solutions are now replaced by concrete models of Cinderella. Each of these aspects has been modeled by one of MATE's model types to demonstrate its effectiveness.

More precisely, the variability within spatial zones and interactions have been modeled as context variability model. An adaptive Petri net specifies, how the robot is expected to operate its workflow depending on the contextual situation. Using stimulus models, the mocked body movement of the human co-worker is controlled from timer transitions that appear in the adaptive Petri net specification. Furthermore, cobotic modes, as proposed by the DLR engineers, have been directly mapped to features of the adaptation mode variability model. Finally, the reconfiguration automaton defines the expected switching behavior between those modes.

In summary, all model types that have been proposed by MATE are relevant to solve the specific requirements of the cobot and self-adaptive application Cinderella. This finding suggests that the introduced concepts are effective and required for ensuring the quality of systems with such degree of self-adaptivity.

7. Summary and Discussion

In this thesis, a model-driven solution to the problem of testing SAS has been proposed. Thus, the overall quality management of autonomous applications, which typically employ self-adaptivity, is simplified. With the introduced concepts, models, and reference architecture in hand, test engineers are equipped to tackle resilience more effiently than with classic model-based testing or plain manual test-case definition. In correspondence to the problems that were stated in Section 1.1, the following solutions have been elaborated:

(S1) A set of **modeling-related concepts and models** have been described in Chapter 4 to overcome problem (P1) of state space explosion. The proposed models provide special expressiveness, which allows for specifying a test generation tool set, respectively a simulator, to explore this state space efficiently. Known and necessary requirements of SAS test approaches, which we worked out of related work in Chapter 3, are all met by this novel approach.

(S2) With the reference architecture of **MATE** in Chapter 5, we have provided a solution to problem (P2) about the need of appropriate tooling for SAS testing so that the conceptional proposals of (S1) can be used technically.

The concepts proposed in this thesis are mainly based on models and, especially when considering test-case generation, the overall approach relates strongly to model-driven software development. From this perspective, the automation of test design is fostered by defining adequate models, which, on the one hand, convey all necessary information, and, on the other hand, are concise enough for managing them efficiently.

The richness of information of the used models has been demonstrated in examples and the experimental study. The models' conciseness is an effect of the amount of automated reasoning performed by the generator or, respectively, simulator. With traditional MBT approaches, a tester would have been required to design a behavioral model of the system for each sequence of contextual changes, which involves the explicit definition of these sequences, manually reasoning about their consequences and defining the expected response of the SASuT. The new model format leads, thus, to an enormous efficiency gain in SAS testing.

However, on the other side, models that are composed and reasoned about automatically make it hard for engineers to keep an overview of the effects of manual changes to the models. For instance, when a test modeler alters the position of a timer transition within an adaptive Petri net, completely different interactions of the SASuT with the generated contexts may be observed. Such correlations are sometimes hard to understand and must be compensated by the tooling that allows for inspecting the results of a specific model change. With MATE, a respective tool chain has been realized, which comprises components to foster such insights. For instance, the

reachability tree can be manually explored, and the reached state of a model instance is visualized right in the editors, where the models were conducted. These built-in mechanisms help engineers to overcome the cognitive gap between modeling and test-case execution.

Static artifacts, namely variability models, mainly span the space of situations of both context and adaptation modes, which relate to each other dynamically. For variability modeling, we found on feature models, which are easy to manage and combine well with value-space classification and boundary analysis from traditional testing. The constraints that a feature tree inherits can completely be mapped to propositional logic, which is used by the interpretation in MATE. An alternative to this approach would be a more mature formalism, such as delta modeling, which allows for further formalizing and thus, controlling, the dynamic change in the definition of context and adaptation.

In conclusion, the introduced models and their implementations not only follow the natural process of defining dynamic contextual influences and, afterward, their impact on a long-running system, but also leverage the ideas from known approaches onto a new level. Instead of only solving a subset of the requirements that authors found for testing SAS, this thesis proposes an integrated method to match all known characteristics.

Furthermore, with the elaborated experimental study on Cinderella in Chapter 6, support for the initially stated hypothesis (cf. Section 1.3) has been demonstrated. The experiment confirms that MATE, with its conceptional background, indeed provides a new, more comprehensive coverage of an SAS' dynamics and its behavioral state space, which now can be tested in an automated fashion. Cinderella not only is an example of a long-running, cooperative, and autonomous system, it also crosscuts the requirements that were found in related work. Both context dynamics and history of the self-adaptive system is relevant for test execution, which would require a large bandwidth of models or a single large model with classic MBT approaches.

7.1. Summary of Contributions

This thesis contributes to the fields of self-adaptive software, to model-based testing research, and variability modeling. Whereas many concepts as feature models, communicating automata, dynamic software products lines, and test-case generation are considered as state of the art, the integration of all those findings is novel. Furthermore, the transitions and the level of conditions to transitions as extensions to Petri nets in combination with externalized functions and timer transitions have not been discussed before in the literature before. Also, the complete modeling process following the counter feedback loop principle is novel to SAS testing.

The briefly announced main contributions of this thesis are:

(C1) **Model-driven methods** for testing before non-covered regions of an SAS' state space based on generated test-cases or simulation in the loop

(C2) A **comprehensive formalization** in the form of metamodels that implement all aspects of the SAS test methods

(C3) A reference architecture for an **integrated test environment** (MATE) that realizes the proposed models and allows for employing them along a standard dynamic test process

Additional to these major ones, several minor contributions are listed in the following:

Minor Contributions in Chapter 4

- **Regarding (C1) Model-based Testing**
  - A conceptional infrastructure to foster the dual use of test-case generation and simulation in the loop

  - Timer transitions as novel concept for communicating between models in a quantitative manner
  - A step-wise extension starting from Petri nets, which allows for employing the concepts and customizing them for the required level of coverage

Minor Contributions in Chapter 5

- **Regarding (C1) Model-Based Testing**
  - A tooling environment, which enables testers to perform all tasks of a standard process of dynamic testing based on the introduced models and architecture
- **Regarding (C2) Comprehensive Formalization**
  - Outgoing from the mathematical formalization in Chapter 4, a re-formalization as object-oriented metamodels is contributed
- **Regarding (C3) Integrated Test Environment**
  - A flexible operator-based framework for extending the models or specifying new stimulus models
  - A test-automation framework for SAS testing
  - An interactive user interface for step-wise simulation of SAS

7.2. Open Research Questions

Despite MATE and its conceptional background provide a more effective solution than earlier approaches, many open questions and future research challenges remain untouched.

One of those unsolved challenges is that future autonomous systems will most probably require real-time features. Autonomous cars, mobile robots, and air vehicles must respond quickly to changes in the environment such as newly detected obstacles, traffic situations or other incidents. The hereby proposed solutions do not provide such real-time mechanisms despite they have been investigated for parts of the used model types, such as timed Petri nets.

For systems with real-time capabilities, also the hereby employed approaches concerning discrete virtual time are insufficient. Continuous sensor input requires a more comprehensive logic.

Another challenging trend is machine learning. Systems that learn do not always create reproducible behavior, which can be tested in advance. On the other side, such learning systems are more effective in unanticipated conditions, which we neglected in this thesis. However, even with such a technology, it would be necessary to limit a respective system in certain behavioral categories, because still person's health and property must be protected. In these cases, MATE can be used to create a larger confidence in an SAS' reliability.

❧

Bibliography

[ABCF11] Uwe Aßmann, Nelly Bencomo, Betty HC Cheng, and Robert B France. Models@ run.time (dagstuhl seminar 11481). *Dagstuhl Reports*, 1(11):91–123, 2011.

[ABZ12] Dhaminda B. Abeywickrama, Nicola Bicocchi, and Franco Zambonelli. SOTA: Towards a General Model for Self-Adaptive Systems. In *IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012*, pages 48–53. IEEE, 2012.

[ACF+09] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, Jean-Paul Rigault, et al. Modeling Context and Dynamic Adaptations with Feature Models. In *Proceedings of the 4th International Workshop Models@ run.time*, 2009.

[ACL+11] Mathieu Acher, Philippe Collet, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling Variability from Requirements to Runtime. *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 77–86, April 2011.

[ADLMW09] Jesper Andersson, Rogerio De Lemos, Sam Malek, and Danny Weyns. Modeling Dimensions of Self-adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems*, pages 27–47. Springer, 2009.

[AHPE07] Mourad Alia, Svein Hallsteinsen, Nearchos Paspallis, and Frank Eliassen. Managing Distributed Adaptation of Mobile Applications. In *Distributed Applications and Interoperable Systems*, pages 104–118. Springer, 2007.

[AHZ13] Dhaminda B. Abeywickrama, Nicklas Hoch, and Franco Zambonelli. SimSOTA: Engineering and Simulating Feedback Loops for Self-Adaptive Systems. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 67–76, New York, NY, USA, 2013. ACM.

[ALRL04] Algirdas Avižienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[ANKH+08] Y. Al-Nashif, A.A. Kumar, S. Hariri, Guangzhi Qu, Yi Luo, and F. Szidarovsky. Multi-Level Intrusion Detection System (ML-IDS). In *International Conference on Autonomic Computing (ICAC)*, pages 131–140, 2008.

[ASP13a] Konstantinos Angelopoulos, Vítor E. Silva Souza, and João Pimentel. Requirements and Architectural Approaches to Adaptive Software Systems: A Comparative Study. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 23–32, Piscataway, NJ, USA, 2013. IEEE Press.

[ASP13b] Konstantinos Angelopoulos, Vítor E Silva Souza, and Joao Pimentel. Requirements and Architectural Approaches to Adaptive Software Systems: A Comparative Study. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 23–32. IEEE Press, 2013.

[Bat05] Don Batory. *Feature Models, Grammars, and Propositional Formulas.* Springer, 2005.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run.time. *Computer*, 42(10), 2009.

[BDG+07] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Bec03] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley Professional, 2003.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *Software Product-Family Engineering*, pages 13–21. Springer, 2002.

[BGT08] Luciano Baresi, Sam Guinea, and Giordano Tamburrelli. Towards Decentralized Self-adaptive Component-based Systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 57–64, New York, NY, USA, 2008. ACM.

[BH06] Tom Broens and Aart Van Halteren. SimuContext: Simply Simulate Context. In *International Conference on Autonomic and Autonomous Systems (ICAS'06)*, pages 45–45, July 2006.

[BHA12] Nelly Bencomo, Svein Hallsteinsen, and Eduardo Almeida. A View of the Landscape of Dynamic Software Product Lines. *Computer*, 2012.

[BM09] Mikael Berndtsson and Jonas Mellin. Eca rules. In *Encyclopedia of Database Systems*, pages 959–960, Boston, MA, 2009. Springer US.

[CdLG+09] BettyH.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, HolgerM. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, HausiA. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In BettyH.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, 2009.

[CDPP96] David M Cohen, Siddhartha R Dalal, Jesse Parelius, and Gardner C Patton. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software*, 13(5):83–88, 1996.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CGFP09] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic Computing Through Reuse of Variability Models at Runtime: The Case of Smart Homes. *Computer*, 42(10):37–43, 2009.

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-Based Feature Modeling and Constraints : A Progress Report. In *Proceedings of the International Workshop on Software Factories*, pages 16–20, 2005.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Dey01] A.K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[DMFM10] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. A Dynamic Software Product Line approach Using Aspect Models at Runtime. In *5th Domain-Specific Aspect Languages Workshop*, 2010.

[dSdL11] Carlos Eduardo da Silva and Rogério de Lemos. Dynamic Plans for Integration Testing of Self-Adaptive Software Systems. In *Proceedings of the 6th international Symposium on Software Engineering for Adaptive and Self-managing Systems - SEAMS '11*, page 148, New York, New York, USA, 2011. ACM Press.

[ESKR14] Benedikt Eberhardinger, Hella Seebach, Alexander Knapp, and Wolfgang Reif. Towards Testing Self-Organizing, Adaptive Systems. In *Testing Software and Systems*, volume 8763 of *Lecture Notes in Computer Science*, pages 180–185. Springer Berlin Heidelberg, 2014.

[FDB+08] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Modeling and Validating Dynamic Adaptation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 97–108. Springer, 2008.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

[GGC+16] A. Giusti, J. Guzzi, D. C. Cire?an, F. L. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. D. Caro, D. Scaramuzza, and L. M. Gambardella. A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. *IEEE Robotics and Automation Letters*, 1(2):661–667, July 2016.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Gre10] Thomas Grechenig. *Softwaretechnik / Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, 2010.

[GSC09] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software Architecture-Based Self-Adaptation. In *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009.

[Hal07] Ibrahim A Halloun. *Modeling Theory in Science Education*, volume 24. Springer Science & Business Media, 2007.

[Hel12a] Michiel Helvensteijn. Dynamic Delta Modeling. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 127–134. ACM, 2012.

[Hel12b] Michiel Helvensteijn. Dynamic Delta Modeling. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 127–134. ACM, 2012.

[HHPS08] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, April 2008.

[HIR03] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Generating Context Management Infrastructure From High-Level Context Models. In *In Proceedings of the 4th International Conference on Mobile Data Management (MDM)-Industrial Track*, 2003.

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 943–944, New York, NY, USA, 2008. ACM.

[Hoa78] Charles Antony Richard Hoare. Communicating Sequential Processes. In *The Origin of Concurrent Programming*, pages 413–443. Springer, 1978.

[Hof98] Douglas Hoffman. A Taxonomy for Test Oracles. In *Quality Week*, volume 98, pages 52–60, 1998.

[Hor01] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology. 2001.

[HSF+11] Sami Haddadin, Michael Suppa, Stefan Fuchs, Tim Bodenmüller, Alin Albu-Schäffer, and Gerd Hirzinger. Towards the Robotic Co-Worker. In *Robotics Research*, pages 261–282. Springer, 2011.

[HSSF06] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using Product Line Techniques to Build Adaptive Systems. In *Proceedings of the Software Product Line Conference, 2006 10th International*, pages 10 pp.–150, 2006.

[IBM06] IBM. An Architectural Blueprint for Autonomic Computing. *IBM White Paper*, 2006.

[IEE13a] Software and Systems Engineering, Software Testing Part 1: Concepts and Definitions. *ISO/IEC/IEEE 29119-1:2013(E)*, pages 1–64, Sept 2013.

[IEE13b] Software and Systems Engineering, Software Testing Part 2: Test Processes. *ISO/IEC/IEEE 29119-2:2013(E)*, pages 1–68, Sept 2013.

[IEE13c] Software and Systems Engineering, Software Testing Part 3: Test Documentation. *ISO/IEC/IEEE 29119-3:2013(E)*, pages 1–138, Sept 2013.

[ISO16] Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models. *ISO/IEC 25010:2011*, pages 1–34, 2016.

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. *Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible*, pages 638–652. Springer Berlin Heidelberg, 2011.

[JZM07] Ke Jiang, Lei Zhang, and S. Miyake. OCL4X: An Action Semantics Language for UML Model Execution. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 633–636, July 2007.

[KACC11] Tariq M King, Andrew A Allen, Rodolfo Cruz, and Peter J Clarke. Safe Runtime Validation of Behavioral Adaptations in Autonomic Software. In *Autonomic and Trusted Computing*, pages 31–46. Springer, 2011.

[KC03] Jeffrey O Kephart and David M Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990.

[KFH15] H. Kopetz, B. Fromel, and O. Hoftberger. Direct Versus Stigmergic Information Flow in Systems-of-Systems. In *10th System of Systems Engineering Conference (SoSE), 2015*, pages 36–41, 2015.

[Kin76] James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KPPR10] Konstantinos Kakousis, Nearchos Paspallis, George Angelos Papadopoulos, and Pedro Antonio Ruiz. ECEASST Testing Self-Adaptive applications with Simulation of Context Events. *Electronic Communications of the EASST*, 28, 2010.

[Lad97] Robert Laddaga. Self Adaptive Software, DARPA Broad Agency Announcement (BAA) 98-12 Proposer Information Pamphlet - Excerpt. http://people.csail.mit.edu/rladdaga/BAA98-12excerpt.html, 1997.

[Lap08] Jean-Claude Laprie. From Dependability to Resilience. In *38th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages G8–G9. Citeseer, 2008.

[LK06] Jaejoon Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *Software Product Line Conference, 2006 10th International*, pages 10–140, 2006.

[LK10] Kwanwoo Lee and Kyo C Kang. Usage Context as Key Driver for Feature Selection. In *Software Product Lines: Going Beyond*, pages 32–46. Springer, 2010.

[LLH05] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using Inter-Procedural Side-effect Information in JIT Optimizations. In *Compiler Construction*, pages 287–304. Springer, 2005.

[LS00] Henry Lieberman and Ted Selker. Out of Context: Computer Systems that Adapt to, and Learn from, Context. *IBM Systems Journal*, 39(3.4):617–632, 2000.

[MA00] Bruno Marre and Agnes Arnould. Test Sequences Generation from Lustre Descriptions: Gatel. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 229–237. IEEE, 2000.

[Mar95] B. Marre. LOFT: A Tool for Assisting Selection of Test Data Sets from Algebraic Specifications. In *Theory and Practice of Software Development*, pages 799–800. Springer, 1995.

[MBS10] André Maaß, Danilo Beucho, and Arnor Solberg. Adaptation Model and Validation Framework Final Version – (DiVA deliverable D4.3), 2010.

[MC11] Radu Muschevici and Dave Clarke. Modular Modelling of Software Product Lines with Feature Nets. In *SEFM 2001 proceedings*, pages 318–333, 2011.

[McC93] John McCarthy. Notes on Formalizing Context. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 555–560. Morgan Kaufmann, 1993.

[MCMO10] Luis Merino, Fernando Caballero, Ivan Maza, and Aníbal Ollero. Automatic Forest Fire Monitoring and Measurement Using Unmanned Aerial vehicles. In *Proceedings of International Conference on Forest Fire Research*. Citeseer, 2010.

[MCP10] Radu Muschevici, Dave Clarke, and J. Proenca. Feature Petri Nets. In *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, volume 2, 2010.

[MDK11] Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai. Application of Software Health Management Techniques. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 1–10, New York, NY, USA, 2011. ACM.

[Mil80] Robin Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, vol. 92, 1980.

[MLSW14] Stephan Mennicke, Malte Lochau, Julia Schroeter, and Tim Winkelmann. Automated Verification of Feature Model Configuration Processes Based on Workflow Petri Nets. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 62–71. ACM, 2014.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[NG15] Stefan Niemczyk and Kurt Geihs. Adaptive Run-Time Models for Groups of Autonomous Robots. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM*, pages 127–133. IEEE, 2015.

[NL13] Kai Nehring and Peter Liggesmeyer. Testing the Reconfiguration of Adaptive Systems. In *Proceedings of The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, pages 14–19. XPS Press, 2013.

[ÖA97] Pinar Öztürk and Agnar Aamodt. Towards a Model of Context for Case-Based Diagnostic Problem Solving. In *Context-97; Proceedings of the Interdisciplinary Conference on Modeling and Using Context*, pages 198–208, 1997.

[OGT+99] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, pages 54–62, 1999.

[OMSM09] Sebastian Oster, Florian Markert, Andy Schürr, and Werner Müller. Integrated Modeling of Software Product Lines with Feature Models and Classification Trees. In *Proceedings of the 1st International Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE 2009)*. CEUR, 2009.

[Par76] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, pages 1–9, 1976.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* Technische Hochschule Darmstadt, 1962.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*, pages 419–443. Springer, 1997.

[RAM+06] Claudia Raibulet, Francesca Arcelli, Stefano Mussino, Mario Riva, Francesco Tisato, and Luigi Ubezio. Components in an Adaptive and QoS-Based Architecture. In *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems*, SEAMS '06, pages 65–71, New York, NY, USA, 2006. ACM.

[Rei05] Stuart Reid. The Art of Software Testing, Second Edition. *Software Testing, Verification and Reliability*, 15(2):136–137, 2005.

[RH04] Venkatesh Prasad Ranganath and John Hatcliff. Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs. In *Compiler Construction*, pages 39–56. Springer, 2004.

[RSPA11] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring Dynamic Software Product Lines. In *ACM SIGPLAN Notices*, volume 47, pages 3–12. ACM, 2011.

[RSTS11] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-Dimensional Variability Modeling. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 11–20. ACM, 2011.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90. IEEE, 1994.

[SBG99] Albrecht Schmidt, Michael Beigl, and Hans-W Gellersen. There is More to Context than Location. *Computers & Graphics*, 23(6):893–901, 1999.

[Sch16] Klaus Schwab. *The Fourth Industrial Revolution.* Crown Business, 2016.

[SLP04] Thomas Strang and Claudia Linnhoff-Popien. A Context Modeling Survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp - Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, May 2009.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*, pages 131–133. Springer-Verlag, Wien, 1973.

[SVGB05] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[Tha13] Bernhard Thalheim. The Conception of the Model. In *Business Information Systems - 16th International Conference, BIS 2013, Poznań, Poland, June 19-21, 2013. Proceedings*, pages 113–124, 2013.

[TKB+14] Thomas Thuem, Christian Keastner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70 – 85, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[TRCPB07] Pablo Trinidad, Antonio Ruiz-Cortés, Joaquín Peña, and David Benavides. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *1st SPLC Workshop on Dynamic Software Product Line (DSPL)*, page 51–56, Kyoto, Japan, 2007.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[WAP] WAPForum. User Agent Profile (UAProf). http://www.wapforum.org, last visisted 02/23/2015.

[WD11] Claas Wilke and Birgit Demuth. UML is Still Inconsistent! How to Improve OCL Constraints in the UML 2.3 Superstructure. In *Proceedings of the International Workshop on OCL and Textual Modelling Colocated with TOOLS Europe 2011, ICMT 2011, TAP 2011 and SC 2011*, 2011.

[WER07] Zhimin Wang, Sebastian Elbaum, and David S Rosenblum. Automated Generation of Context-aware Tests. In *Proceedings of the 29th International Conference on Software Engineering, 2007. ICSE 2007.*, pages 406–415. IEEE, 2007.

[ZD11] Christopher Zhong and Scott A. DeLoach. Runtime Models for Automatic Reorganization of Multi-robot Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 20–29, New York, NY, USA, 2011. ACM.

[ZH00] Hong Zhu and Xudong He. A Theory of Testing High Level Petri Nets. In *Proceedings of International Conference on Software in Theory and Practice, IFIP World Computer Congress*, pages 21–25. Citeseer, 2000.

[ZH14] Gefei Zhang and Matthias Hölzl. A Set of Metrics for States and Transitions in UML State Machines. In *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*, BM-FA '14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.

APPENDICES

Cinderella Definitions

1. Cinderella Adaptation Bounds

```
[
 { "currentState": -1, "transitions": [
  { "from": -1, "to": 0, "moveable": "GripperReal", "isEnter": true},
  { "from": 0, "to": -1, "moveable": "GripperReal", "isEnter": false},
  { "from": -1, "to": 1, "moveable": "hand", "isEnter": true},
  { "from": 1, "to": -1, "moveable": "hand", "isEnter": false},
  { "from": 0, "to": 2, "moveable": "hand", "isEnter": true},
  { "from": 2, "to": 0, "moveable": "hand", "isEnter": false},
  { "from": 1, "to": 2, "moveable": "GripperReal", "isEnter": true},
  { "from": 2, "to": 1, "moveable": "GripperReal", "isEnter": false}
  ] ,
  "maxPoint": { "x": 1000.0, "y": -250.0, "z": 300.0 },
  "minPoint": { "x": 300.0, "y": -550.0, "z": -500.0 },
  "name": "srcA"
 },
 { "currentState": -1, "transitions": [
  { "from": -1, "to": 0, "moveable": "GripperReal", "isEnter": true},
  { "from": 0, "to": -1, "moveable": "GripperReal", "isEnter": false},
  { "from": -1, "to": 1, "moveable": "hand", "isEnter": true},
  { "from": 1, "to": -1, "moveable": "hand", "isEnter": false},
  { "from": 0, "to": 2, "moveable": "hand", "isEnter": true},
  { "from": 2, "to": 0, "moveable": "hand", "isEnter": false},
  { "from": 1, "to": 2, "moveable": "GripperReal", "isEnter": true},
  { "from": 2, "to": 1, "moveable": "GripperReal", "isEnter": false}
   ] ,
  "maxPoint": { "x": 1000.0, "y": 0.0, "z": 300.0 },
  "minPoint": { "x": 300.0, "y": -250.0, "z": -500.0 },
  "name": "good"
 },
 { "currentState": -1, "transitions": [
  { "from": -1, "to": 0, "moveable": "GripperReal", "isEnter": true},
  { "from": 0, "to": -1, "moveable": "GripperReal", "isEnter": false},
  { "from": -1, "to": 1, "moveable": "hand", "isEnter": true},
  { "from": 1, "to": -1, "moveable": "hand", "isEnter": false},
  { "from": 0, "to": 2, "moveable": "hand", "isEnter": true},
  { "from": 2, "to": 0, "moveable": "hand", "isEnter": false},
  { "from": 1, "to": 2, "moveable": "GripperReal", "isEnter": true},
  { "from": 2, "to": 1, "moveable": "GripperReal", "isEnter": false}
   ] ,
  "maxPoint": { "x": 1000.0, "y": 250.0, "z": 300.0 },
  "minPoint": { "x": 300.0, "y": 0.0, "z": -500.0 },
  "name": "bad"
 },
 { "currentState": -1, "transitions": [
  { "from": -1, "to": 0, "moveable": "GripperReal", "isEnter": true},
  { "from": 0, "to": -1, "moveable": "GripperReal", "isEnter": false},
  { "from": -1, "to": 1, "moveable": "hand", "isEnter": true},
  { "from": 1, "to": -1, "moveable": "hand", "isEnter": false},
```

```
  { "from": 0, "to": 2, "moveable": "hand", "isEnter": true},
  { "from": 2, "to": 0, "moveable": "hand", "isEnter": false},
  { "from": 1, "to": 2, "moveable": "GripperReal", "isEnter": true},
  { "from": 2, "to": 1, "moveable": "GripperReal", "isEnter": false}
   ] ,
  "maxPoint": { "x": 1000.0, "y": 550.0, "z": 300.0 },
  "minPoint": { "x": 300.0, "y": 250.0, "z": -500.0 },
  "name": "srcB"
 },
 { "currentState": -1, "transitions": [
  { "from": -1, "to": 0, "moveable": "center", "isEnter": true,
   "commands" [
    { "type": "org.tud.inf.st.iotfog.adapt.SpeedCmd", "instance": { "speed": 0.3 } }
   ] },
  { "from": 0, "to": -1, "moveable": "center", "isEnter": false,
   "commands" [
    { "type": "org.tud.inf.st.iotfog.adapt.SpeedCmd", "instance": { "speed": 1.0 } }
   ] }
   ] ,
  "maxPoint": { "x": 3000.0, "y": 1500.0, "z": 1500.0 },
  "minPoint": { "x": -400.0, "y": -1500.0, "z": -1500.0 },
  "name": "awareness"
 }
]
```

Listing 1: bounds-cinderella.js

2. Cinderella Self-adaptive Workflow

```
{ "current": 0,
 "tasks": [
  { "name": "c0", "posture": "center.js", "steppable": false },
  { "name": "c1", "posture": "centersrca.js", "steppable": false },
  { "name": "c2", "posture": "center.js", "steppable": false },
  { "name": "c3", "posture": "centersrca.js", "steppable": false },
  { "name": "c4", "posture": "centersrcb.js", "steppable": false },
  { "name": "c5", "posture": "center.js", "steppable": false },
  { "name": "a0", "posture": "srca.js", "steppable": true },
  { "name": "a1", "posture": "srca.js", "steppable": true },
  { "name": "b0", "posture": "srcb.js", "steppable": true },
  { "name": "b1", "posture": "srcb.js", "steppable": true },
  { "name": "g0", "posture": "snka.js", "steppable": true },
  { "name": "g1", "posture": "snka.js", "steppable": true },
  { "name": "g2", "posture": "snka.js", "steppable": true },
  { "name": "g3", "posture": "snka.js", "steppable": true },
  { "name": "x0", "posture": "snkb.js", "steppable": true },
  { "name": "x1", "posture": "snkb.js", "steppable": true },
  { "name": "x2", "posture": "snkb.js", "steppable": true },
  { "name": "x3", "posture": "snkb.js", "steppable": true }
 ],

 "edges": [
  { "from": 0, "to": 6, "condition":
   { "boundsRef": "srcA", "state": 1, "negation": true, "conjunction":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
     { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
      { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
       { "boundsRef": "bad", "state": 2, "negation": true } } } } } },
  { "from": 6, "to": 1, "condition":
   { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
       { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
       { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
       { "boundsRef": "bad", "state": 2, "negation": true } } } } },
  { "from": 1, "to": 10, "condition":
   { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
       { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
       { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
       { "boundsRef": "bad", "state": 2, "negation": true } } } } },
  { "from": 1, "to": 14, "condition":
```

```
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 10, "to": 2, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 14, "to": 2, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 2, "to": 7, "condition":
    { "boundsRef": "srcB", "state": 1, "negation": false, "conjunction":
        { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } } },
   { "from": 2, "to": 8, "condition":
    { "boundsRef": "srcB", "state": 1, "negation": true, "conjunction":
        { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } } },
   { "from": 7, "to": 3, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 3, "to": 11, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 3, "to": 15, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 11, "to": 2, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 15, "to": 2, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 8, "to": 5, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 5, "to": 12, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 5, "to": 16, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 12, "to": 0, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 16, "to": 0, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
```

```
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 0, "to": 9, "condition":
    { "boundsRef": "srcA", "state": 1, "negation": false, "conjunction":
        { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } } },
   { "from": 9, "to": 4, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 4, "to": 13, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 4, "to": 17, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 13, "to": 0, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } },
   { "from": 17, "to": 0, "condition":
    { "boundsRef": "srcA", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "srcB", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "good", "state": 2, "negation": true, "conjunction":
        { "boundsRef": "bad", "state": 2, "negation": true } } } } }
 ]
}
```

Listing 2: test.js